

FREENIX Track

2001 USENIX Annual Technical Conference

*Boston, Massachusetts, USA
June 25–30, 2001*

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: <http://www.usenix.org>

The price is \$22 for members and \$30 for nonmembers.
Outside the U.S.A. and Canada, please add
\$18 per copy for postage (via air printed matter).

Past FREENIX Proceedings

FREENIX '00	2000	San Diego, CA	\$22/30
FREENIX '99	1999	Monterey, CA	\$22/30

© 2001 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-10-3

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

**Proceedings of the
FREENIX Track**

2001 USENIX Annual Technical Conference

**June 25–30, 2001
Boston, Massachusetts, USA**

Program Organizers

Program Chair

Clem Cole, *Paceline Systems Corp.*

Program Committee

Ken Coar, *The Apache Software Foundation/IBM*

Chris Demetriou, *Broadcom Corp*

Ted Faber, *ISI/USC*

Drew Gallatin, *Duke/FreeBSD*

Alan Nemeth, *Compaq*

Simon Patience, *Zambeel Inc*

Garry Paxinos, *Metro Link/XFree86*

Stephen Tweedie, *Red Hat*

The USENIX Association Staff

External Reviewers

Jim Bound, *Nokia*

Wayne Cardoza, *Compaq*

Doug Cassell, *Compaq*

TW Cook, *Tower Technology*

Carl Davidson, *HP*

Miguel deIcaza, *Helix Code/The Gnome Foundation*

Gerry Feldman, *Boston Linux Users*

Peter Honeyman, *University of Michigan/OpenBSD*

Jim McGinness, *BSDi*

Wendy Rannenberg, *Compaq*

Ralph Swick, *World Wide Web Consortium/MIT*

Jonathan Ward, *Compaq*

Jonathan Wistar, *Vividon*

Erez Zadok, *SUNY at Stony Brook*

FREENIX Track

2001 USENIX Annual Technical Conference

June 25-30, 2001

Boston, Massachusetts, USA

Index of Authors	vi
Message from the Program Chair	vii

Thursday, June 28

Mac Security

Session Chair: Simon Patience, Zambeel Inc.

LOMAC: MAC You Can Live With	1
<i>Timothy Fraser, NAI Labs</i>	

TrustedBSD: Adding Trusted Operating System Features to FreeBSD	15
<i>Robert N. M. Watson, FreeBSD Project, NAI Labs</i>	

Integrating Flexible Support for Security Policies into the Linux Operating System	29
<i>Peter Loscocco, NSA; and Stephen Smalley, NAI Labs</i>	

Scripting

Session Chair: Erez Zadok, SUNY at Stony Brook

A Practical Scripting Environment for Mobile Devices	43
<i>Brian Ward, University of Chicago</i>	

Nickle: Language Principles and Pragmatics	55
<i>Bart Massey, Portland State University; and Keith Packard, SuSE Inc.</i>	

The Design and Implementation of the NetBSD rc.d System	69
<i>Luke Mewburn, Wasabi Systems, Inc.</i>	

User Space

Session Chair: Alan Nemeth, Compaq

User-Level Checkpointing for LinuxThreads Programs	81
<i>William R. Dieter and James E. Lumpp, Jr., University of Kentucky</i>	

Building an Open-source Solaris-compatible Threads Library	93
<i>John Wood, Compaq Computer UK Ltd</i>	

Are Mallocs Free of Fragmentation?	105
<i>Aniruddha Bohra, Rutgers University; and Eran Gabber, Lucent Technologies-Bell Labs</i>	

Friday, June 29

User Environment

Session Chair: Ken Coar, The Apache Software Foundation/IBM

- Sandboxing Applications119
Vassilis Prevelakis, University of Pennsylvania; and Diomidis Spinellis, Athens University
- Building a Secure Web Browser127
Sotiris Ioannidis, University of Pennsylvania; and Steven M. Bellovin, AT&T Labs–Research
- Citrus Project: True Multilingual Support for BSD Operating Systems135
Jun-ichiro itojun Hagino, Internet Initiative Japan Inc.

Kernel

Session Chair: Drew Gallatin, Duke/FreeBSD

- Kqueue—A Generic and Scalable Event Notification Facility141
Jonathan Lemon, FreeBSD Project
- Improving the FreeBSD SMP Implementation155
Greg Lehey, IBM LTC Ozlabs
- Page Replacement in Linux 2.4 Memory Management165
Rik van Riel, Conectiva Inc.

Storage

Session Chair: Clem Cole, Paceline Systems Corp.

- User-Level Extensibility in the Mona File System173
Paul W. Schermerhorn, Robert J. Minerick, Peter Rijks, and Vincent W. Freeh, University of Notre Dame
- Volume Managers in Linux185
David Teigland and Heinz Mauelshagen, Sistina Software, Inc.
- The Design and Implementation of a Transparent Cryptographic File System for UNIX199
Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano, Università di Salerno

Saturday, June 30

Graphics

Session Chair: Garry Paxinos, Metro Link/XFree86

- Design and Implementation of the X Rendering Extension213
Keith Packard, XFree86 Core Team, SuSE Inc.
- Scwm: An Extensible Constraint-Enabled Window Manager225
Greg J. Badros, InfoSpace.com; Jeffrey Nichols, Carnegie Mellon University; and Alan Borning, University of Washington
- The X Resize and Rotate Extension—RandR235
Jim Gettys, Compaq; and Keith Packard, XFree86 Core Team, SuSE Inc.

Securing Networks

Session Chair: Ted Faber, ISI/USC

MEF: Malicious Email Filter—A UNIX Mail Filter That Detects Malicious Windows Executables245
Matthew G. Schultz and Eleazar Eskin, Columbia University; Erez Zadok, SUNY Stony Brook; Manasi Bhattacharyya and Salvatore J. Stolfo, Columbia University

Cost Effective Security for Small Businesses253
Sean R. Brown, Applied Geographics, Inc.

Heimdal and Windows 2000 Kerberos—How to Get Them to Play Together267
Assar Westerlund, Swedish Institute of Computer Science; and Johan Danielsson, Center for Parallel Computers, KTH

Resource Management

Session Chair: Theodore Ts'o, VA Linux Systems

Predictable Management of System Resources for Linux273
Mansoor Alicherry, Bell Labs; and K. Gopinath, Indian Institute of Science

Scalable Linux Scheduling285
Stephen Molloy and Peter Honeyman, CITI—University of Michigan

A Universal Dynamic Trace for Linux and Other Operating Systems297
Richard J. Moore, IBM, Linux Technology Centre

Index of Authors

Alicherry, Mansoor	273	Lumpp Jr., James E.	81
Badros, Greg J.	225	Massey, Bart	55
Bellovin, Steven M.	127	Mauelshagen, Heinz	185
Bhattacharyya, Manasi	245	Mewburn, Luke	69
Bohra, Aniruddha	105	Minerick, Robert J.	173
Borning, Alan	225	Molloy, Stephen	285
Brown, Sean R.	253	Moore, Richard J.	297
Cattaneo, Giuseppe	199	Nichols, Jeffrey	225
Catuogno, Luigi	199	Packard, Keith	55, 213, 235
Danielsson, Johan	267	Persiano, Pino	199
Del Sorbo, Aniello	199	Prevelakis, Vassilis	119
Dieter, William R.	81	Riel, Rik van	165
Eskin, Eleazar	245	Rijks, Peter	173
Fraser, Timothy	1	Schultz, Matthew G.	245
Freeh, Vincent W.	173	Schermerhorn, Paul W.	173
Gabber, Eran	105	Smalley, Stephen	29
Gettys, Jim	235	Spinellis, Diomidis	119
Gopinath, K.	273	Stolfo, Salvatore J.	245
Hagino, Jun-ichiro itojun	135	Teigland, David	185
Honeyman, Peter	285	Ward, Brian	43
Ioannidis, Sotiris	127	Watson, Robert N. M.	15
Lehey, Greg	155	Westerlund, Assar	267
Lemon, Jonathan	141	Wood, John	93
Loscocco, Peter	29	Zadok, Erez	245

Message from the Program Chair

In the over 20 years that I have been part of this community, one of my greatest satisfactions has been participating in the changes the organization has taken on as the community has grown. USENIX itself has always been about change. The core UNIX technology represents a sea of change from its predecessors. Now, as UNIX moves toward middle age, we as a community are embroiled in another change, popularly known as the Freely Available Software or Open Source movement.

The concept behind this movement is of course even older than USENIX. In fact, the founding ideas behind USENIX map directly into the Freely Available Software movement. USENIX was one of the original places where, as computer scientists, we could share our ideas, our code, and our technology in an open and free manner.

Four years ago, USENIX made a bold stroke. The leaders of the organization had noted that across the globe, the free software movement had developed a great deal of good, but largely unpublished, work. As a professional organization, USENIX wished to become a magnet for these developers and often first-time authors. Therefore USENIX added to its Annual Technical Conference a parallel track, the "FREENIX track."

Building FREENIX has not been easy, yet the enthusiasm and interest continue to grow. 2001 is the second straight year of a formal peer-review process and upholds the standard set last year: every FREENIX presentation has a corresponding completed paper in these proceedings!

These papers and this conference did not happen by accident. Last November, FREENIX received 58 submissions to its Web site. Every member of the Program Committee read each submission, and all were read and commented on by external reviewers familiar with the topic.

In January, we met as a committee for two days to choose from among the submissions the 27 that were to become the full papers you find in this book. Each paper was assigned a shepherd who helped the author polish his or her submission into the paper you see here. Our goal was to put together a balanced program, not merely to focus on issues traditional from the BSD or Linux kernel world. We are quite pleased both with the breadth of subject range of the submissions and with the diversity of the final program.

As the Program Chair, I wish to thank all the authors for their hard work. Furthermore, I cannot say "thank you" often enough to the Committee and the reviewers. Their hours spent reading submissions and drafts bring us, the final readers, these fine results. I have enjoyed working with them and getting to know them all a little better. Also I cannot forget the wonderful staff in the USENIX office--without their support, guidance, and wisdom, none of this would ever happen!

I hope you who make it to the Annual Technical Conference in Boston will spend time in the FREENIX sessions, and those of you who could not attend this year, I hope to see at the next FREENIX. Either way, as readers of these proceedings, we all can learn from these wonderful and exciting works coming from the community of Freely Available Software.

Clement T. Cole, FREENIX Program Chair

LOMAC: MAC You Can Live With

Timothy Fraser NAI Labs
tfraser@nai.com 3060 Washington Road
Glenwood, MD 21738

Abstract

LOMAC is a security enhancement for Linux kernels. LOMAC demonstrates that it is possible to apply Mandatory Access Control techniques to standard Linux kernels already deployed in the field, and to do so in a manner that is simple, compatible, and largely invisible to the traditional Linux user. The LOMAC Loadable Kernel Module protects the integrity of critical system processes and files from viruses, worms, Trojan horses, and malicious remote users. It is compatible with standard Linux 2.2 kernels and applications, and seeks to provide useful protection without site-specific configuration. LOMAC is designed to be a form of MAC that typical users can live with.

1 Introduction

Over the last 25 years, many projects have demonstrated useful Mandatory Access Control (MAC) features on UNIX systems. Two early examples include KSOS [18] and UCLA Secure UNIX [23]. More recent examples include DTE [3], and Security-Enhanced Linux [17]. However, despite their success, these demonstrations have not prompted widespread adoption of MAC in mainstream UNIX kernels.

One likely explanation for this lack of widespread adoption may be overall cost of use: In these demonstrations, the new MAC features came at the cost of incompatibility with existing kernel and application software, increased administrative overhead, or a disruption of traditional usage patterns. Among typical users, the overall cost of adopting the new MAC features outweighed the perceived benefits, discouraging widespread mainstream adoption.

The LOMAC project is an attempt to bring simple but useful MAC integrity protection to Linux in a form that:

- is applicable to standard kernels,
- is compatible with existing applications,
- requires no site-specific configuration, and
- is largely invisible to traditional users.

In short, LOMAC aims to provide a form of MAC that typical users can live with [19]. LOMAC implements a form of Low Water-Mark MAC integrity protection [5] in a Loadable Kernel Module (LKM). Administrators can load the LOMAC LKM into standard, off-the-CD-ROM Linux 2.2 kernels, including both kernels distributed in binary form and kernels built from standard sources. Once loaded, the LOMAC LKM protects the integrity of critical system processes and files from viruses, worms, Trojan horses, and malicious remote users. Because of its compatible design, LOMAC can be used to provide integrity protection for presently-deployed systems based on standard Linux kernels with little impact on their normal operation.

Several theoretical aspects of the LOMAC project have been discussed in a previous paper [9]. These aspects include LOMAC's application of Low Water-Mark model, the UNIX compatibility benefits of models like Low Water-Mark over many better-known models, and some of the drawbacks of LOMAC's LKM-based implementation with regard to the reference monitor approach [1]. This paper, on the other hand, will focus on the details of LOMAC's implementation, paying particular attention to the techniques required to enhance standard Linux kernels without patching their source, and to manage security attributes without kernel and filesystem support.

The discussion begins with section 2, which describes the integrity protection provided by LOMAC. This is followed by a detailed examination of LOMAC's architecture and implementation in section 3, focusing on LOMAC's use of interposition and implicit attribute mapping to maintain compatibility with standard Linux kernels. Section 4 explains how LOMAC applies its

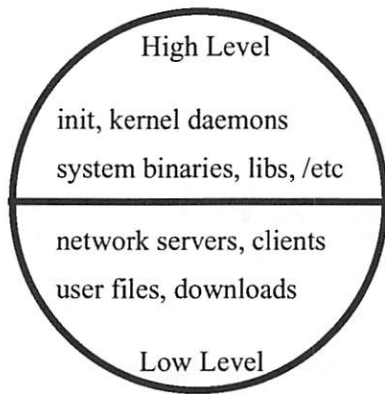


Figure 1: LOMAC's 2-level partitioning of a system.

protection mechanism in a manner that encourages application compatibility and avoids administrative overhead. Section 5 presents the results of some performance benchmarks, and discusses potential optimizations. Section 6 addresses usability concerns and lists some future directions for LOMAC, including strategies to overcome some of its present shortcomings and an upcoming port to FreeBSD. Section 7 follows with a summary of related efforts to enhance the security of Linux kernels. Finally, section 8 presents some conclusions.

2 Protection

LOMAC provides protection by dividing a system into two integrity levels: high and low. The diagram in figure 1 illustrates this division. The high level contains critical system components that must be protected, such as the init process, kernel daemons, system binaries, libraries and configuration files. The low level contains the remaining components, such as client and server processes that read from the network, local user processes and their files. Once LOMAC assigns a file to one level or the other, its level never changes. This is not so for processes: LOMAC can “demote” high-level processes by reducing their levels to low during run-time. LOMAC never increases the level of a process. Section 4 describes how LOMAC decides which files and processes belong in which part; this section summarizes how LOMAC uses this division to provide protection.

When LOMAC is running, a process's level determines how much power it has to modify other parts of the system. Given the above division of the system into two levels, LOMAC provides integrity protection with two

main mechanisms. First, LOMAC prevents low-level processes from modifying (writing, truncating, deleting) high-level files or signalling high-level processes. Since non-administrative users, their network clients, and all network servers run at the low level, these restrictions protect the high-level part of the system from direct attacks by malicious remote users and compromised servers.

Second, LOMAC ensures that (potentially dangerous) data does not flow from low-level files to high-level files. A process could attempt to cause such a flow by reading from a low-level file (as data or as program text) and subsequently writing to a high-level file. LOMAC prevents such flows through demotion: whenever a high-level process reads from a low-level file, LOMAC reduces the process's level to low. Once at the low integrity level, LOMAC's first mechanism prevents the process from modifying high-level files, as described above. This combination of mechanisms prevents indirect attacks by viruses, worms and Trojan horses.

LOMAC cannot distinguish between a program that has read low-integrity data but is still running properly and one that has read low-integrity data and has been compromised. However, LOMAC *can* ensure that processes which read potentially dangerous low-level data during run-time are demoted to the low integrity level. Once at this low level, LOMAC's other mechanisms prevent them from harming high-integrity processes or files.

3 Implementation

There are two main problems in implementing kernel-resident MAC: gaining supervisory control over kernel operations, and mapping security attributes to files. There are a range of potential solutions to these problems, each embodying a different tradeoff between features such as generality and efficiency, and costs such as incompatibility with existing software and the need for configuration. LOMAC has chosen low cost solutions in both cases. LOMAC uses interposition at the kernel's system call interface [10, 11, 20] to gain supervisory control. LOMAC uses implicit attribute mapping [3] to map security attributes to files. These choices may not be as supportive of generality and efficiency as alternate approaches involving direct modifications of the kernel source. However, they allow LOMAC to operate on standard Linux kernels already deployed in the field - an essential part of LOMAC's approach to encouraging adoption.

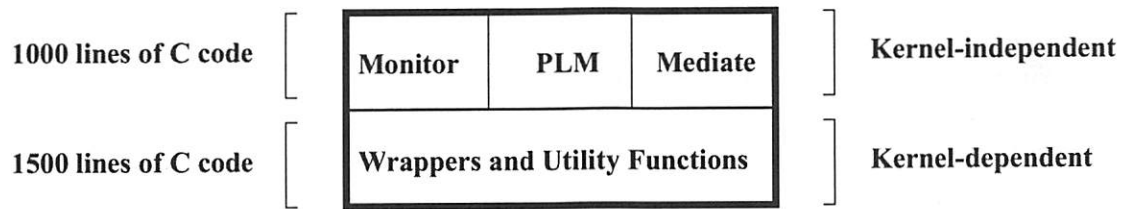


Figure 2: LOMAC Loadable Kernel Module Architecture

Figure 2 shows the architecture of the LOMAC LKM. The diagram shows a horizontal split between upper and lower halves. The upper half implements high-level LOMAC functionality in a kernel-independent manner, and consists of approximately 1000 lines of C code (counting only those lines containing semicolons or braces). The lower half implements a kernel-specific interface to the Linux 2.2 series of kernels, and consists of approximately 1500 lines of C code. An alternate Linux 2.0 interface was supported in the past; alternate Linux 2.4 and FreeBSD interfaces are expected in the future.

3.1 Gaining control

In order to provide protection, LOMAC must gain supervisory control over kernel operations - that is, LOMAC must be able to make access control decisions as described in section 2, and compel the kernel to enforce them. LOMAC achieves this control by interposing itself between processes and the kernel at the kernel's system call interface. LOMAC's kernel interface contains a series of functions called "wrappers," due to their similarity to Generic Software Wrappers [10]. Ultimately, there will be one such wrapper for each security-relevant Linux system call; some wrappers have not yet been implemented in the present version of LOMAC. Each wrapper takes the same parameters as its corresponding system call. At initialization time, LOMAC traverses the kernel's system call vector, which is essentially an array of function pointers through which the kernel provides services to user processes. LOMAC replaces the addresses of security-relevant system calls with the addresses of the corresponding wrappers. Once done, calls made through the system call vector will call the wrappers, rather than the kernel's corresponding system call functions.

Wrappers follow the algorithm shown in figure 3. First, LOMAC performs mediation: it decides whether to allow or deny the calling process's request for service. It bases this decision on a comparison of the calling pro-

cess's level and the levels of the arguments, as described in section 2. If LOMAC decides to deny, it returns an appropriate error code to the caller. Otherwise, LOMAC proceeds to the next step, where it calls the kernel's original system call function to provide the actual service. Finally, LOMAC monitors the completion of the kernel's original system call, updating its data structures to reflect changes in the system state. This is where LOMAC demotes processes, and marks the in-memory data structures representing open files (dentry structures) with the appropriate levels for future reference.

Viewed from a high level of abstraction, this interposition-based wrapper algorithm is not overly complex. However, implementing it in a manner that avoids Time-Of-Check, Time-Of-Use (TOCTOU) errors requires care [11, 26]. Early versions of LOMAC had many TOCTOU errors: Wrappers would copy user-space pathname arguments into kernel-space, and make mediation decisions based on these copies. After positive decisions, the kernel's original system call functions would copy the pathnames into kernel-space a second time, and operate on this second copy. The potential existed for a user process to make a system call with an allowable pathname and change it to a non-allowable pathname after LOMAC had made its mediation decision, but before it called the kernel's original system call function. This ability to change pathnames between the

```

wrapper( arguments ) {
    Mediate: decide to allow
              or deny the operation;

    call kernel's original
    system call function;

    Monitor: update LOMAC's state
              on successful completion;
}

```

Figure 3: Wrapper Algorithm

```

01: int wrap_open( const char *filename, int flags, int mode ) {
02:     char *k_filename_s, *k_canabspath_s;
03:     struct dentry *p_dentry, *p_dir_dentry;
04:     struct file *p_file;
05:     int ret_val;
06:
07:     if( IS_ERR( ( k_filename_s = getname( filename ) ) ) ) {
08:         return( PTR_ERR( k_filename_s ) );
09:     }
10:     if( !( k_canabspath_s = (char *)__get_free_page( GFP_KERNEL ) ) ) {
11:         ret_val = -ENOMEM;
12:         goto out_putname;
13:     }
14:     if( ( ret_val = make_canabspath( k_filename_s, k_canabspath_s,
15:         &p_dir_dentry, &p_dentry ) ) ) {
16:         goto out_dputs;
17:     }
18:
19:     if( ( flags & O_TRUNC ) ||
20:         ( ( flags & O_CREAT ) && ( !p_dentry ) ) ) {
21:         if( !( p_dentry && WRITE_EXEMPT( p_dentry ) ) ) {
22:             if( !( mediate_subject_object("open",current,p_dir_dentry) ) ) {
23:                 ret_val = -EACCES;
24:                 goto out_dputs;
25:             }
26:             if( !( mediate_subject_path("open",current,k_canabspath_s) ) ) {
27:                 ret_val = -EACCES;
28:                 goto out_dputs;
29:             }
30:         } /* if this is not an exempt case */
31:     } /* if we should mediate */
32:
33:     TURN_ARG_CHECKS_OFF;
34:     ret_val = ((int (*)(const char *, int, int))orig_open)
35:         ( k_canabspath_s, flags, mode );
36:     TURN_ARG_CHECKS_ON;
37:     if( ret_val >= 0 ) {
38:         p_file = fget( ret_val );
39:         monitor_open( current, p_file->f_dentry );
40:         fput( p_file );
41:     }
42:
43: out_dputs:
44:     if( p_dir_dentry ) { dput( p_dir_dentry ); }
45:     if( p_dentry ) { dput( p_dentry ); }
46:     free_page( (unsigned long)k_canabspath_s );
47: out_putname:
48:     putname( k_filename_s );
49:     return( ret_val );
50: } /* wrap_open() */

```

Figure 4: C source for LOMAC v1.1.0's wrapper for sys_open (run-time assertions and most comments removed).

time of LOMAC's check, and the time the kernel used the pathname gave user processes the opportunity to defeat LOMAC's protection.

Figure 4 illustrates the solution to the TOCTOU problem: copy pathname arguments into kernel-space at the beginning of the wrapper, and invoke the kernel's original system call with the address of this copy, rather than the address of the original user-space buffer. The figure contains the C source for LOMAC's open system call (`sys_open`) wrapper. The source shows the additional buffer-copying, as well as the unusual toggling of the Linux kernel's sense of the kernel-/user-space boundary required to make the its original system calls accept these copies.

In its first 18 lines, the wrapper examines its arguments, gathering the information it needs in later steps. Line 7 copies the filename argument into kernel-space to avoid TOCTOU errors. All subsequent operations are on this copy, rather than the user-space original. LOMAC determines the levels of files based on their absolute canonical-form pathnames using an algorithm discussed in the next subsection. Line 14 prepares the filename for its level determination by converting it into this form.

The nested `if` statements in lines 19 through 21 ensure that LOMAC performs mediation only when there is the potential for a file creation or truncation. LOMAC does not mediate writes to files in the open wrapper. This mediation is handled by other wrappers corresponding to the Linux kernel's various write system calls. The `WRITE_EXEMPT` macro on line 21 exists to allow harmless truncates of device special files such as serial lines and terminals. Similar exemptions exist in the write system call wrappers. These exemptions allow low-level processes to perform I/O on these devices, while keeping the device special files themselves in the high-level part of the system.

Lines 22 through 32 perform the actual mediation. Before allowing the open, LOMAC makes checks both on the file and on its parent directory, as traditional UNIX does. Line 22 ensures that the calling process has sufficiently high integrity level to modify the contents of the named file's parent directory. Line 26 ensures that the calling process has a sufficiently high integrity level to create or truncate the named file. These checks are handled by functions in the kernel-independent part of the LOMAC LKM.

Lines 33 through 36 invoke the kernel's original system call function using the wrapper's kernel-space copy of

the filename argument. When serving user processes, the kernel's system calls expect to copy their pathname arguments from user-space. Before copying, the system calls execute a check to ensure that the pathname buffer address is indeed on the user side of the kernel-/user-space boundary - a check that will normally fail on the wrapper's kernel-space pathname buffers. Fortunately, the kernel provides a mechanism to disable this check on a per-process basis. The macros on lines 33 and 36 toggle this check off for the duration of the original system call function. For safety, the canonical-absolute pathname conversion function on line 14 performs the safety checks that LOMAC turns off in the original system call.

Lines 37 through 50 conclude the wrapper. If the open system call succeeded in opening a file, lines 37 through 41 call LOMAC's kernel-independent open monitoring function to label the file's in-memory data structure (`dentry`) with the appropriate level. The various read and write wrappers will subsequently use this label when they mediate and monitor operations on the file.

As shown in figure 4, it takes a considerable amount of wrapper code to support mediation and monitoring in an interposition-based scheme. The extra buffer copy to avoid TOCTOU errors adds overhead. Similarly, many wrappers contain nested `if` statements like those in lines 19 through 22 to predict, based on the arguments, what operation the kernel will eventually perform. The read and write wrappers require more extensive logic, because these system calls must handle operations on a variety of objects (files, pipes, sockets), each of which requires different mediation and monitoring.

An alternative approach to gaining control might be to patch the kernel source, placing mediation and monitoring further down in the kernel, at the point closer to where it operates on objects. This move would reduce overhead by eliminating the extra TOCTOU buffer copies and the need to predict the kernel's behavior ahead of time. However, this patching strategy is not presently an option for LOMAC, which must avoid modifying kernel source in order to maintain compatibility with existing kernels.

3.2 Attribute Mapping

In addition to gaining supervisory control, LOMAC must also assign integrity levels to files in a manner that is persistent across reboots. LOMAC maintains a persistent mapping between levels and absolute canonical pathnames in its Path Level Map (PLM) module.

<i>level</i>	<i>flags</i>	<i>path</i>
high		"/home/httpd"
low	child-of	"/home"
high		"/"

Table 1: Three Path-Level Map Rules

Whenever the kernel opens a file, LOMAC labels its in-memory data structure (`dentry`) with the integrity level indicated by the PLM.

LOMAC's PLM implements a simple form of implicit attribute mapping [3]. Given an absolute canonical pathname, it consults a data structure similar to the abridged one shown in table 1. This data structure is an array of records, each a level, flag, path triplet. The records are sorted, longest path first. The basic algorithm is, given a target path, its level can be found by searching linearly through the list of records until a record is found whose path is a prefix of the target path. The level in this record is the proper level for the file named by the target path. For example, the level of `"/home/httpd/html"` is high, because it matches the record for prefix `"/home/httpd"`. The attribute mapping is "implicit" because the appropriate level of a large number of files is implied by a small set of rules.

The child-of flag adds a slight bit of additional complexity. For example, the list of records uses the child-of flag in the record for `/home`. This record indicates that all children of `/home` are low by default. Because of the child-of flag, the record does not apply to `/home` itself, only its children.

If, during a search through the record list, the target path matches a record's path exactly, the flag field is checked. If the child-of flag is set, the match is ignored, and the search continues. Consequently, the level of `"/home/httpd"` is high because it exactly matches the record for prefix `"/home/httpd"`, which has no child-of flag. The level of `"/home/tfraser"` is low because it matches the record for prefix `"/home"` with the child-of flag, and the level of `"/home"` is high because it skips the child-of `"/home"` record and matches the record for prefix `"/"`.

The actual list of PLM records used by the present version of LOMAC contains 25 records. The PLM can map levels to files on any type of filesystem, including remote network filesystems. It requires no filesystem support for storing attributes on disk. Since the PLM's list of rules is completely static, it is trivially persistent across

reboots, and is not susceptible to consistency problems if the filesystem is modified while LOMAC is not running.

The PLM does have two main drawbacks, however. First, it requires canonical absolute pathnames as input. Determining the canonical absolute form of a pathname in a system call wrapper adds overhead.

Second, the PLM can produce inconsistent integrity level results when queried on files named by multiple hard links: If the different hard link names correspond to different levels, the PLM will return whichever level corresponds to the hard link name specified in a query. LOMAC prevents the creation of such confusing hard links during its run-time; administrators must take care to avoid creating them before they load LOMAC. This problem does not extend to symbolic links. LOMAC calls the appropriate kernel functions to translate all paths into canonical (all symbolic links translated) absolute (relative to the root directory) form before examining them. Consequently, LOMAC handles symbolic links properly.

4 Application

In order to apply the protection scheme described in section 2, LOMAC must be able to determine the appropriate level for every process and file in the system. This section describes how LOMAC makes this determination. LOMAC's choice of solution impacts both application compatibility and the degree to which LOMAC remains invisible to users. It is also essential to LOMAC's ability to automatically assign the appropriate levels to users and network servers without site-specific configuration.

4.1 Dividing the Filesystem

Section 3.2 explained how LOMAC uses a small set of rules to determine which parts of the filesystem are at the high integrity level, and which are at the low level. These rules are presently set at compile-time. Although future versions of LOMAC may provide a more configurable rule set, the goal of the present implementation is to deliver a single generic configuration that provides at least some protection on a wide variety of systems.

The division described by the current rule set reflects the tension between two competing goals: providing

the maximum amount of protection, and maintaining the maximum amount of application compatibility. The first goal demands that all files be at the high level, where LOMAC will keep them safe from modification by low-level processes. However, the second goal demands that all files be at the low level, where LOMAC will never prevent low-level processes from modifying them. This second goal is important to compatibility - preventing file modifications can introduce incompatibilities by causing applications to fail.

LOMAC's present division is a compromise between these goals that emphasizes application compatibility. The division roughly parallels the traditional UNIX boundary between the portion of the filesystem owned by the root user (high), and the portion owned by local non-root users (low). This parallelism helps to reduce LOMAC's visibility to non-root users. For example, LOMAC tends to prevent the same operations as the traditional UNIX access control mechanisms: high-level files tend to be owned by the root user. Non-root user processes run at the low level. LOMAC prevents low-level processes from modifying high-level files. However, this behavior is often not surprising because the familiar UNIX access controls would also prevent these modifications as attempted non-root modifications of root-owned files. Only when a low-level process acquires root privileges does the difference become readily apparent - a low-level root process has greatly reduced powers in the presence of LOMAC.

4.2 Monitoring Processes

While file levels are static, process levels can decrease during run-time. In general, LOMAC assigns a new process the same level as the process who created it. At initialization time, LOMAC assigns the high integrity level to the first process (the `idle/init` process), which initializes the system by creating a new high-level process to handle various system tasks. These processes continue by creating more high-level children. As individual processes read from low-level files, LOMAC demotes them to the low integrity level. From that point on, all their children begin life at the low integrity level.

This demotion behavior allows LOMAC to automatically assign user sessions to the appropriate level. For example, with a console login, the `init`, `getty`, and login processes all run at a high level. Upon verifying a user's identity, login spawns a child which executes the user's shell. The shells of non-root users immediately read resource files from the low-level part of the system, caus-

ing LOMAC to demote them. From that point on, their children operate at a low level. LOMAC does not demote the root user's shell because the root user's home directory and its contents are at a high level. The root user's shell may therefore create high-level children, although LOMAC will demote them if they go on to read from the low-level part of the system. This automatic assignment of levels allows LOMAC to provide protection without being configured to recognize a site-specific list of users.

LOMAC also uses its demotion behavior to automatically confine programs that use the network to interact with (potentially malicious) remote entities. LOMAC treats all network interfaces as low-level files. As soon as a process reads from a network interface, LOMAC demotes it to the low integrity level. This scheme places network clients and servers at a safe, low level at the moment they first risk compromise - that is, as soon as they receive their first communication from the network. Furthermore, this scheme allows LOMAC to provide protection without being configured to recognize a site-specific list of potentially dangerous network-readers - LOMAC simply waits for a potentially dangerous network read operation and then makes the appropriate demotion.

4.3 Exceptions for Compatibility

LOMAC's protection scheme is specifically designed to prevent possibly malicious remote entities from using the network to command local processes to modify local `/etc` configuration files. Unfortunately, this scenario essentially describes the purpose of `pump`, the client-side DHCP agent: `pump` modifies local configuration files such as `/etc/resolv.conf` on behalf of remote DHCP servers. Similarly, LOMAC's protection scheme is specifically designed to prevent processes from transferring data from low-integrity to high-integrity files. Unfortunately, this is essentially what occurs as log messages travel from low-integrity processes to the high-integrity system log file through the system log daemon, `syslogd`.

In both these cases, LOMAC must make an exception to allow these critical programs to operate properly. To this end, LOMAC maintains a short list of "trusted" programs. LOMAC never demotes processes that are running trusted programs. Being free from demotion, as long as `pump` and `syslogd` begin running at a high level, they will remain at that level and operate properly. Since trust frees a program only from LOMAC's demo-

tion behavior, running a trusted program at the low integrity level does not provide any additional privileges. Still, the presence of trusted programs represents some risk. If a high-level process running a trusted program were compromised, LOMAC would not prevent it from harming the high-integrity part of the system.

LOMAC also uses the trusted program mechanism to make some concessions to usability. Because it demotes network-reading programs, LOMAC effectively prevents remote administration. (A level-1 process cannot modify critical configuration files, even with the root identity.) Since remote administration is critical to some real-world operations, LOMAC trusts the Secure Shell daemon `sshd`. This arrangement grants administrators high-level user sessions via SSH, as follows:

LOMAC demotes untrusted remote login daemons such as `telnetd` and `rlogind` as soon as they read from the network, preventing them from forking off high-level children. However, because of LOMAC's trust, high-level processes running `sshd` can read from the network without being demoted, and fork off high-level processes to run local user shells. With the trusted `sshd` acting as an un-demotable bridge to the network interface, these local user shells escape demotion themselves by interacting with the network only indirectly, through high-level pseudoterminal devices.

LOMAC also provides a trusted file upgrader, `lup`. When run at a high integrity level, `lup` allows administrators to copy low-integrity files (such as downloaded software updates) to the high-integrity area of the system, presumably after manually verifying that they represent no threat to integrity. The `lup` program is effectively a limited version of `cp` with additional logging. Only its trust-enabled escape from demotion allows it to upgrade files. Consequently, running `lup` from a low integrity level will not permit a user to write a high-level file.

4.4 LOMAC and root

Although LOMAC's division of the system attempts to parallel the traditional UNIX root/non-root boundary for the sake of compatibility, LOMAC's protection mechanism does not depend on the Linux kernel's existing root-identity-based protection mechanism. LOMAC provides protection by observing requests for service made by processes at the kernel's system call interface, and denying those requests it identifies as threats to the integrity of the system. It is not aware of the Linux no-

tion of user identity; consequently it does not allow the root user any special privileges. Conversely, LOMAC does not override, disable, or weaken the existing Linux protection mechanisms: When LOMAC is running, an operation will be allowed if and only if both LOMAC and the existing Linux protection mechanisms agree it should be allowed.

Since LOMAC's strategy of controlling the transfer of data is orthogonal to the traditional UNIX root-based mechanism, it is also orthogonal to efforts to increase the granularity of this root-based mechanism, such as `Linux-privs` [21].

5 Performance

Table 2 shows the results of three benchmarks comparing the performance of Linux kernels running with ("LOMAC v1.1.0") and without LOMAC ("No LOMAC"). The benchmarks tested version 1.1.0 of LOMAC with run-time assertions disabled. The first entry in the table measure the time to perform the "make" portion of the Linux 2.2.5 kernel build procedure on 450MHz Intel Pentium II-based RedHat 6.0 system. Each result is the average of 10 trials, discarding an initial uncounted trial to prime caches. Although this macro-benchmark tends to hide LOMAC's additional kernel overhead, it gives an impression of how a user might perceive LOMAC's performance on a real workload.

The second and third table entries show the latency and throughput performance of the Apache/1.3.9 web server running on a 133MHz Intel Pentium-based RedHat 6.1 system. This web server was connected via a 10Mbit crossover (uplink) Ethernet cable to a Sun Microsystems Ultra 5 workstation running Solaris 2.6. This workstation performed a series of 47 10-minute-long trials running the WebStone 2.5b4 web server benchmark using 32 test clients applying the standard WebStone static workload to the webserver to produce each result. The apparent small improvement in latency is spurious; the performance impact of LOMAC is much smaller than the variance in the WebStone benchmark's results.

The remaining table entries show the results of the BYTE UNIX benchmarks performed with the UnixBench 4.1.0 software on the same system used for the kernel-build benchmark. Each result is the average of 21 trials. The table omits the largely computational DhryStone and WhetStone components of the bench-

Kernel Build Elapsed Time (s)			
	mean	std. dev.	penalty
No LOMAC	269.61	0.03	-
LOMAC v1.1.0	278.05	0.03	3.1%
Webstone Latency (s)			
	mean	std. dev.	penalty
No LOMAC	0.569	0.003	-
LOMAC v1.1.0	0.567	0.003	-0.2%
Webstone Throughput (Mbit/s)			
	mean	std. dev.	penalty
No LOMAC	8.327	0.058	-
LOMAC v1.1.0	8.305	0.063	0.3%
UB Exec1 Throughput (loops/s)			
	mean	std. dev.	penalty
No LOMAC	642.4	23.7	-
LOMAC v1.1.0	537.0	21.7	16.4%
UB File Copy 256 Byte buffers (KByte/s)			
	mean	std. dev.	penalty
No LOMAC	34393	289	-
LOMAC v1.1.0	31131	222	9.5%
UB File Copy 1024 Byte buffers (KByte/s)			
	mean	std. dev.	penalty
No LOMAC	69672	385	-
LOMAC v1.1.0	66155	573	5.0%
UB File Copy 4096 Byte buffers (KByte/s)			
	mean	std. dev.	penalty
No LOMAC	81379	547	-
LOMAC v1.1.0	79078	775	2.8%
UB Pipe Throughput (loops/s)			
	mean	std. dev.	penalty
No LOMAC	263124	1679	-
LOMAC v1.1.0	234225	4289	11.0%
UB Pipe-based Context Switch (loops/s)			
	mean	std. dev.	penalty
No LOMAC	139917	1827	-
LOMAC v1.1.0	116993	1510	16.4%
UB Process Creation (loops/s)			
	mean	std. dev.	penalty
No LOMAC	3811	20	-
LOMAC v1.1.0	3830	24	-0.5%
UB System Call Overhead (loops/s)			
	mean	std. dev.	penalty
No LOMAC	249414	332	-
LOMAC v1.1.0	249356	303	0.2%
UB 8 Shell Script Load (loops/minute)			
	mean	std. dev.	penalty
No LOMAC	144.2	3.0	-
LOMAC v1.1.0	129.0	3.1	10.5%

Table 2: Benchmark Results

mark; the presence of LOMAC did not significantly affect these components. The apparent small improvement in process creation time is also spurious; the performance impact of LOMAC is smaller than the variance in the Process Creation portion of the UnixBench benchmark.

LOMAC's performance is comparable to interposition-based general kernel extension mechanisms such as Generic Software Wrappers [10] and SLIC [11]. For example, the SLIC prototype reported performance penalties ranging from 0% to 5% on an emacs-building benchmark, depending on how many security extensions were loaded at the time. The Generic Software Wrappers prototype reported penalties ranging from 3.5% to 6.5% on a kernel-building benchmark, up to 1.4% for WebStone latency, and up to 3.3% for WebStone throughput, again depending on how many security extensions were loaded.

LOMAC has not yet been optimized for performance; there are several areas of its implementation that trade performance for simplicity in order to support the rapid development of new features. For example, when a process opens or executes a file, LOMAC consults the PLM to determine the file's level and the level of its parent directory. LOMAC saves these levels in memory for the benefit of its read and write mediation functions. However, LOMAC makes no attempt to skip the PLM lookup on subsequent opens, even for files and directories that already have their levels stored in memory. The PLM implementation is presently based on a simple but inefficient sequential search. Lookups on short, common directories such as "/bin" and "/usr/bin" require 25 string comparisons. This inefficiency is reflected in the high penalty shown by the UnixBench Exec1 Throughput benchmark. Considerable time could be saved by avoiding redundant PLM lookups, and by improving the PLM's search algorithm.

At a higher level, LOMAC might save time by not mediating the actions of high-level processes, since LOMAC always allows high-level processes to do as they wish. Similarly, LOMAC might save time by not considering low-level processes for demotion, since low-level processes are already running at the lowest integrity level. This optimization has the potential to reduce the overhead of read and write operations shown in the three UnixBench File Copy benchmarks. As LOMAC nears its goals for features, an increasing amount of development resources will be allocated to improving performance.

6 Discussion

Section 4 presents an analytical argument for the usability of LOMAC, describing how LOMAC is designed to be compatible with existing applications, and is largely invisible to non-root users. Although there have been no formal usability studies of LOMAC, there is some anecdotal evidence of its compatibility with traditional UNIX: In order to test LOMAC under normal usage conditions, NAI Labs' Chief Scientist runs LOMAC on his Linux workstation. However, he was forced to turn LOMAC off near the end of January, 2001 while the author fixed a serious bug. On the evening of 31 January 2001, the author completed the fix and re-installed LOMAC on the chief scientist's workstation. Significantly, he carelessly forgot to inform anyone of what he had done. LOMAC was not discovered until 11 days later, when the author mentioned the re-installation in casual conversation. Although the chief scientist's overall usage of the workstation during that period was light, the fact remains that LOMAC was sufficiently compatible with traditional UNIX to remain undetected by an highly experienced user until it was unwittingly revealed by its author.

There is much work yet to be done on LOMAC. With more development, LOMAC can overcome many of its present limitations. The remainder of this section summarizes some possible future directions.

Improved handling of "/tmp": In its present state, the PLM prevents the effective use of temporary files by high-level processes. Directories like "/tmp" must be able to contain files of different integrity levels where the appropriate level can be determined only by considering the level of the file's creator, not by considering its pathname. The PLM presently supports only low-level files in "/tmp", making it impossible to run temporary-file-dependent programs like emacs or gcc at a high level.

There are at least two ways in which the PLM might be extended to overcome this problem. The PLM might be extended to polyinstantiate "/tmp", providing separate temporary directories for each level in a manner that is transparent to processes. Alternately, the PLM might apply a new flag to "/tmp" indicating that the levels of files there should be based on the level of the creating process. Both of these solutions involve tradeoffs: A polyinstantiated "/tmp" may confuse users ("why can't my low-level process see that high-level temporary file?").

On the other hand, allowing files to inherit their creators' levels will add complexity - the present invariant that a file's level may always be determined by its pathname greatly simplifies many aspects of the LOMAC code.

Complete controls: LOMAC does not yet control all critical kernel operations. For example, even though LOMAC controls the kernel's read and write system calls, processes may still bypass LOMAC by modifying files via memory-mapping. Access to memory-mapped files is difficult for LOMAC to mediate because once a process maps a file, it may modify the file through memory operations that do not require system calls. To solve this problem, LOMAC might perform pessimistic read/write mediation at the time a file is memory-mapped, and revoke or downgrade dangerous mappings upon process demotion. Several other kernel abstractions also lack sufficient controls, including message queues, semaphores, and all forms of shared memory.

Port to Linux 2.4, FreeBSD, TrustedBSD: As was described in section 3, LOMAC's architecture includes a separate kernel-dependent interface. Although earlier versions of LOMAC had alternate interfaces for Linux 2.0 and 2.2, only the 2.2 interface is supported in the present version. The 2.2 interface does not support the 2.4 Linux kernel; a new interface will be required. Experience with these interfaces has shown that LOMAC tracks changes in the Linux kernel relatively easily: because it has so few dependencies on the kernel source, porting has been required only between major kernel revisions (2.x, not 2.2.x) so far.

An interposition-based port to FreeBSD is scheduled for the second half of 2001. As the TrustedBSD project begins to provide improved kernel support interfaces for LKMs like LOMAC, the author will port to these interfaces, as well. In addition to making LOMAC available to more users, these ports will provide an opportunity to reimplement LOMAC's kernel interfaces with the benefit of previous experience. These implementations may provide better performance and additional features, such as multiprocessor support.

Improved confinement: LOMAC protects the integrity of high-level processes and files, but does not provide any protection for the low-level part of the system. For example, although LOMAC prevents a compromised low-level server from installing trapdoors and Trojan horses in the high-level part of

<i>project</i>	<i>patch</i>	<i>module</i>	<i>general wrappers</i>	<i>access control</i>	<i>intrusion detection</i>
Beattie MAC	x			x	
Generic Software Wrappers		x	x		
Immunix/SubDomain	x	x		x	x
Janus (Linux)		x		x	
Kernel Hypervisors		x	x		
LIDS	x			x	x
LOMAC		x		x	
Medusa DS9	x	x		x	
Pitbull LX		x		x	
RSBAC	x			x	
SAIC DTE	x			x	
SELinux	x			x	
VXE	x			x	
William&Mary DTE	x			x	

Table 3: A Comparison of Related Projects

the system, it does not prevent a compromised low-level server from harming the integrity of the low-level part of the system, perhaps by destroying low-integrity user files, or by sending kill signals to other low-level servers. This drawback was due to the manner in which the Low Water-Mark model divides a system “horizontally” into levels, separating only high from low. Lipner has suggested an enhancement that would add additional “vertical” divisions, separating one server from another within a given level [16]. The potential of this technique to improve LOMAC remains to be explored.

Configurable levels: Early versions of LOMAC supported configurations with more than two levels, and allowed administrators to assign different levels to each network interface. One useful three-level configuration worked well on a host with two network interfaces: The configuration placed system objects and processes at the highest level, most local user processes, most user files and an interface to an internal network at the middle level, and the remaining servers and an interface to the an external network at the lowest level. This three-level configuration provided integrity protection to a larger portion of the system than LOMAC’s present two-level configuration by bringing some user resources into the upper two protected levels.

However, some user files had to remain at the unprotected lowest level where programs that read from the lowest-level network interface, such as E-mail agents and web browsers, could modify them. Consequently, the three-level configuration was more visible to non-root users than the two-

level configuration, because it forced them to operate at multiple levels. For example, it forced users to run separate text editor processes for modifying lowest-level and middle-level files, and to choose the proper editor depending on the situation.

Because this complexity conflicts with LOMAC’s emphasis on remaining largely invisible to the user, this functionality has not yet been carried forward into the present prototype. However, future versions of LOMAC might be extended to allow site-specific configurations with many levels, and offer the existing two-level configuration as a default. In a configuration with many levels, support for programs that are trusted only in restricted ranges of levels may also be useful [15].

7 Related work

There are a wide variety of projects aimed at improving the security of Linux kernels using interposition and/or MAC. Examples include Generic Software Wrappers [10], the recent Linux port of Janus [12], Kernel Hypervisors [20], LIDS [28], Malcolm Beattie’s MAC [4], Medusa DS9 [29], Pitbull LX [2], RSBAC [22], SAIC DTE [24], Security-Enhanced Linux [17], Immunix/Subdomain [7], VXE [14], and William&Mary DTE [13].

Different projects emphasize different goals. Table 3 compares these projects according to several criteria. The first two criteria deal with implementation: Those

projects that modify the kernel source receive a mark in the *patch* column, those that use an LKM receive a mark in the *module* column. The last three criteria deal with features: Projects that seek to provide general support for kernel security extension through system call interposition receive a mark in the *general wrappers* column. Those that provide MAC functionality are marked in the *access control* column. Finally, those that provide or are bundled with other useful security functionality, such as intrusion detection, are marked in the *intrusion detection* column. The projects that have the most relevance to LOMAC's goal of encouraging adoption by decreasing the overall cost of use are the four that avoid modifying kernel source.

Of these projects, Generic Software Wrappers and Kernel Hypervisors seek to provide general support for kernel extensions. Conceivably, LOMAC could be implemented in the frameworks they provide. The remaining two, Pitbull LX and Janus, attempt only to implement a single form of MAC, as LOMAC does. Pitbull LX and Janus provide protection by confining potentially dangerous applications according to the principle of Least Privilege [25]. They lessen their impact on UNIX compatibility by confining only certain applications, rather than applying their controls to every process on the system.

Each of these four LKM-based approaches has the potential to provide a very small overall cost of use, particularly if they were distributed in a form that lessened administrative overhead and did not overly disrupt typical usage patterns,

In the wake of the 2001 Linux Kernel Summit, several organizations have begun efforts to improve the Linux kernel's support for security enhancements like LOMAC. The TrustedBSD project [27] is also developing similar improvements for the FreeBSD kernel. The aspect of these efforts that is most relevant to LOMAC is their plan to provide a new means of gaining supervisory control over kernel operations. The Linux efforts are concentrating on placing "hooks" at strategic points inside the kernel. These hooks will transfer control to security modules like LOMAC, allowing them to make access control decisions. It is reasonable to expect a future version of LOMAC based on these hooks to perform better than the present one; placing the hooks inside the kernel has the potential to eliminate the need for much of the operation-prediction and buffer-copying overhead imposed by interposition at the system call interface.

Fortunately, LOMAC's architecture has strong separation between the interposition-based interface and the

rest of the LOMAC LKM. When such hooks become standard kernel features, this separation will allow LOMAC to discard its present interposition-based interface and make use of them.

8 Conclusions

LOMAC's present implementation shows that it is possible to apply Mandatory Access Control techniques to standard off-the-CD-ROM Linux kernels. LOMAC uses interposition at the system call interface to gain supervisory control over kernel operations, and implicit attribute mapping to mark files with persistent labels.

By confining network-reading applications to the low integrity level, LOMAC prevents compromised servers, worms, and malicious remote users from harming the integrity of the high-level part of the system, even when they have root privilege. By demoting processes that read or execute low-integrity data, LOMAC ensures that network-imported virus and Trojan horse programs will be similarly confined, even if they are initially read or executed by root-privileged high-level processes. Due to this confinement, such malicious programs cannot copy themselves or be copied by others into the high-integrity part of the system.

Furthermore, LOMAC's protection scheme requires no support from applications. LOMAC's access control functionality is automatic: Applications do not need to request that it be applied. It is also transparent: LOMAC interposes itself at the kernel's system call interface, requiring only the standard parameters, and returning only the standard error codes. LOMAC does not require users or applications to explicitly choose roles [8] or domains [6]. Because of the automatic and transparent nature of its protection mechanism, LOMAC can operate with existing applications, even those distributed in binary-only form.

LOMAC is designed to be compatible with existing software, largely invisible to traditional Linux users, and applicable without site-specific configuration. In short, it is designed to be a form of MAC that typical users can live with.

9 Acknowledgements

The author would like to thank Lee Badger for his long-standing support for the LOMAC project, and for suggesting the title of this paper.

LOMAC is Free software available for download under the GNU General Public License from <ftp://ftp.tislabs.com/pub/lomac>.

References

- [1] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.
- [2] Argus Systems Inc. Pitbull LX. http://www.argus-systems.com/products/white_paper/lx/.
- [3] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A Domain and Type Enforcement UNIX Prototype. *USENIX Computing Systems*, 9(1):47–83, Winter 1996.
- [4] M. Beattie. MAC. <http://users.ox.ac.uk/~mbeattie/linux>.
- [5] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.
- [6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, September 1985.
- [7] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, December 2000.
- [8] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Baltimore, Maryland, October 1992.
- [9] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Berkeley, California, May 2000.
- [10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Berkeley, California, May 1999.
- [11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.
- [12] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, California, July 1996.
- [13] S. Hallyn and P. Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, Georgia, October 2000.
- [14] U. InteS, Odessa. VXE. <http://www.intes.odessa.ua/vxe>.
- [15] T. M. P. Lee. Using Mandatory Integrity to Enforce “Commercial” Security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 140–146, Oakland, California, April 1988.
- [16] S. B. Lipner. Non-Discretionary Controls for Commercial Applications. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 2–10, Oakland, California, April 1982.
- [17] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001.
- [18] E. J. McCauley and P. J. Drongowski. KSOS – The Design of a Secure Operating System. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 345–353, Montvale, New Jersey, 1979.
- [19] M. D. McIlroy and J. A. Reeds. Multilevel security with fewer fetters. In *Proceedings of the USENIX UNIX Security Workshop*, pages 24–31, August 1988.
- [20] T. Mitchem, R. Lu, and R. O’Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the 13th Annual Computer Security Applications Conference*, San Diego, California, December 1997.
- [21] A. G. Morgan. linux-privs. <http://www.kernel.org/pub/linux/libs/security/linux-privs/old/doc>.
- [22] A. Ott. Regel-basierte Zugriffskontrolle nach dem Generalized Framework for Access Control-Ansatz am Beispiel Linux. Master’s thesis, Universitat Hamburg, Fachbereich Informatik, 1997.
- [23] G. J. Popek, M. Kampe, C. S. Kline, A. Stoughton, M. Urban, and E. J. Walton. UCLA Secure UNIX. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 355–364, Montvale, New Jersey, 1979.
- [24] SAIC. SAIC DTE. <http://research-cistw.saic.com/cace/dte.html>.
- [25] J. H. Saltzer and M. D. Schroder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE Vol. 63(9)*, pages 1278–1308, September 1975.
- [26] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, DC, August 1999.
- [27] R. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001.
- [28] H. Xie and P. Biondi. LIDS. <http://www.lids.org>.
- [29] M. Zelem, M. Pikula, and M. Ockajak. Medusa DS9. <http://medusa.fornax.sk>.

TrustedBSD

Adding Trusted Operating System Features to FreeBSD

Robert N. M. Watson
FreeBSD Project, NAI Labs
rwatson@{FreeBSD.org,tislabs.com}

Abstract

Trusted operating systems provide a “next level” of system security, offering both new security features and higher assurance that they are properly implemented. TrustedBSD is an on-going project to integrate a number of trusted OS features into the open source FreeBSD operating system, and involves both architectural and development process improvements. This paper describes how the open source development practices of the FreeBSD Project impacted the design and implementation choices for these features, and describes lessons learned that will influence future work. Several key TrustedBSD features are discussed as examples of how new security services may be introduced in such an environment.

1 Introduction

TrustedBSD[20][22] is a project to add trusted operating system functionality to FreeBSD[7][14], including improvements to the kernel and userland security infrastructure, services to better support security features, and specific security features including Access Control Lists (ACLs), fine-grained privileges (“Capabilities”), and Mandatory Access Control (MAC). While TrustedBSD is still under development, several features are already complete and now integrated into the base FreeBSD distribution for inclusion in FreeBSD 5.0. This process has resulted in a great deal of gained experience, which in turn can be used to draw useful conclusions about how future work, especially with regards to

development practices, should be performed. The integration of new security features in FreeBSD offers a number of practical lessons, both technical and social.

This paper introduces the basic features that make up TrustedBSD, describes the goals and processes by which these are being accomplished, details a subset of the features, and reflects on lessons learned as well as future directions for work.

2 TrustedBSD Feature Set

The TrustedBSD feature set attempts to address a number of requirements originating from several communities. This includes the traditional trusted operating system community, in the form of the Orange Book[16], and more recently the Common Criteria[5][11][10], but also the more widespread desire in the FreeBSD community for improved security functionality in the form of greater protection flexibility. These features improve both improvements in existing code, development process improvements, and specific new features.

- *Improved security consistency and correctness.* An important part of introducing new security models and extensively modifying security subsystems is verifying that they are correctly implemented. The FreeBSD developer community did not have test suites covering security behavior, so tests are being developed to determine that no new vulnerabilities are introduced, and so that changes can be experimentally quantified.

An early observation made during this process was that supposedly-equivalent security checks would often be implemented differently. For example, different access control checks were used for the two ways in which debugging can be attached to a process, `ptrace()` and the process file system. Little or no code sharing, combined with an incremental development style, has lead to inconsistent and undocumented protection behavior, especially in the areas of inter-process authorization and file system permission evaluation.

Correctness is clearly an important part in any security project; however, without appropriate tools to verify correctness, it can be difficult to achieve. Access control consistency (and hence abstraction), careful documentation, and extensive and rigorous testing are necessary to accomplish this goal.

- *Improved security abstractions and modularity.* Introducing common implementations of access control check code across check instances improves consistency, but also allows the introduction of improved abstractions, making it easier to introduce new models by substituting policy logic at well-defined enforcement points. Likewise, improving the abstractions associated with the labeling of subjects (process credentials) and system objects (such as files, network packets and interfaces, and kernel management services) allows the more consistent introduction of comprehensive security features such as many MAC policies.
- *General services to support security requirements.* A number of the components of TrustedBSD have usefulness outside of the base TrustedBSD feature set, providing utility for purposes other than purely security work. One example of this is in file system Extended Attributes (EAs), which provide a general interface and implementation for associating arbitrary meta-data with files and directories. A number of the new security services require that additional security labels be associated with file system objects; however, EAs can also be used

by applications to store version meta-data, portability information, or even file icons.

- *Fine-grained Discretionary Access Control (DAC).* The UNIX permission model allows users to specify discretionary protections for objects they have created; this mechanism, however, is inflexible and inexpressive, particular in large-scale environments, or where there are complex security requirements.

POSIX.1e[9] Access Control Lists (ACLs) allow object owners to specify finer granularity protections for file system objects. The ACL implementation leverages the availability of a general EA service, and provides a high level of compatibility with the permission model. ACLs are not only a relatively simple implementation task, but are also a “bullet feature” expected by many operating system consumers, making them a particularly appealing target.

- *Fine-grained privilege model.* One traditional criticism of the UNIX security model has been its reliance on a single concentration of system privileges in the “root” user. This focuses an unnecessarily high level of privilege in a large number of applications, violating the principle of least privilege, and leaving the applications as easy targets for attackers.

POSIX.1e capabilities decompose the root privilege set into several logical components, decoupling privilege from the UID of the process. Processes may manage the availability, inheritance, and effectiveness of capabilities, limiting the scope of damage due to compromise. This implementation leverages EAs to bind capabilities to binaries, and improved security abstractions to replace the superuser access control checks.

- *Mandatory Access Control (MAC).* MAC permits security administrators to define mandatory security policies regarding the relationships between subjects and objects in the system. Traditional MAC policies have included Multi-Level Security (MLS)[3] which provides a military-style confidentiality policy, as well as a variety of integrity and safety models

such as Biba integrity models[4], compartmentalization models, and more general policy mechanisms such as TE[13] and DTE[2].

Free UNIX-like systems have traditionally lacked such features, which can provide higher levels of protection; mandatory policy enforcement is one of the determining features associated with the traditional trusted operating system. Many of the enforcement points for MAC already exist in FreeBSD by virtue of the existing security models, including the Jail[12] model, but improved labeling and access control abstractions, as well as the ability to store labels persistently in EAs, are required for most MAC policies.

- *Plugability.* A long-term architectural goal is to allow the rapid introduction of new security services by continuing to improve abstractions and encouraging modularity. An important element of this is untangling the existing set of security models into independently structured components which are then cleanly composed to generate access control decisions. Providing an easy means to introduce and integrate new models will promote the development of new security features by simplifying the development process. This goal presents substantial challenges, both to design and implement, and in maintaining the necessary performance and usability characteristics.
- *Documentation and Education.* The TrustedBSD feature set introduces a large number of new interfaces and services that are relevant to both system developers and users. A substantial and continuing effort is being invested in maintaining up-to-date developer documentation, and as features are integrated back into the base FreeBSD distribution, user documentation must also be brought up-to-date. Trusted operating system features are a topic unfamiliar to many system programmers and users: detailed yet readable documentation is vital to both maintaining the correctness of the implementation, and to introduc-

ing operating system consumers to the suite of newly available functionality.

3 Implementation and Process Goals

FreeBSD is an actively developed and widely-deployed high performance production operating system. As a result, introducing new security features into the base distribution places a number of constraints on their implementation. This includes the desire to introduce features that provide a rapid security benefit, to avoid degrading performance in existing deployed configurations, to introduce features in a manner compatible with the FreeBSD development and release schedule, and to participate in on-going education and public relations to introduce and promote the ideas and features within the FreeBSD community. In addition, outreach to other operating system communities is necessary to develop standards for application interfaces so as to assure portability of applications taking advantage of these features.

3.1 Satisfying the Requirements of a General-purpose Operating System

- *Rapid security improvement.* Target improvements that have a demonstrable security impact in the short term without high development cost, such as improved access control consistency, code sharing, correctness checking. These changes produce real-world security improvements, as well as making it easier and safer for developers to integrate new security models.
- *Popular features first.* Target features that are more generally useful first, especially primitives that may be reused. This includes extended attributes, ACLs, and improvements to existing security services such as the `jail()` code. Some features, such as MAC, offer benefits but require a substantially higher investment, as well as further research into how

they can be deployed in existing environments, and should be considered long-term goals.

- *Minimize cost on today's deployed installations.* Initially optimize for minimal impact on currently deployed configurations (where new security features will not be enabled), maintaining performance and ease of use. Otherwise, the community will resist the efforts to introduce new features, as they would be contrary to stated goals of the FreeBSD project. Integrating slower features into the base distribution as optional components will, however, increase exposure in the broader developer community increasing the chances of additional developers picking up and working on the implementation to improve perceived performance problems.
- *Support the applications.* Provide security services that allow as many existing applications to run as possible. In practice, this has two implications: first, applications unaware of security mechanisms must behave correctly, and if they must fail, do so safely, and second, security-aware applications must continue to function properly, possibly adapting to support new security primitives. This strategy encourages the adoption of new security features while avoiding introducing risks by changing the fundamental assumptions of application interfaces (in particular, POSIX).

3.2 Emphasis on Portability

Each TrustedBSD feature has introduced a plethora of new APIs for providing access to new services. If similar services exist on other platforms, it is desirable that applications written on one platform be portable to the other. This will be possible only through close communication and cooperation with other vendors, and in some cases, through the development of new standards.

The starting point for this work has been POSIX.1e, a withdrawn IEEE specification draft intended to provide portable interfaces

for Access Control Lists, Auditing, Capabilities, Information Labeling, and Mandatory Access Control. As many of these topics are contentious within the security community, large parts of the draft are effectively unusable as they constitute a consensus on the need for a feature, rather than practical interface details needed for actual implementation. However, the ACL and Capability components of the draft are quite usable, with partial implementations of both widespread. We selected Draft 17, the final draft of the specification, as a starting point. We made extensions or modifications where necessary to disambiguate aspects of the draft, provide functionality not anticipated by the draft writers, or to handle non-POSIX and BSD-specific extensions.

POSIX.1e does not describe extended attributes (EAs), although a number of POSIX.1e implementations rely on EAs to provide storage to support its features. This includes SGI's Trusted Irix[18], FreeBSD, and now also Linux[8]. As EAs will likely be consumed by applications directly, as well as by kernel security services, adopting consistent application interface syntax and semantics is highly desirable. The POSIX.1e online discussion mailing list has provided a forum for the discussion of EA interfaces; a final interface has not been agreed upon, but there is a reasonable consensus on the desired semantics.

The mandatory access control interface described in POSIX.1e, on the other hand, may be too specific to the MLS and Biba MAC models, which each define a dominance operator, requiring a policy that orders labels. The interface also lacks a means by which user processes can host objects and enforcement points, but rely on the operating system to provide label management and policy service. There is substantial consensus in the broader community that more general access control primitives are required to support a broad array of flexible policy mechanisms, and that the POSIX.1e interfaces may provide a useful starting point for that work.

Working with existing models, where possible, offers substantial benefits in the form of application portability. It also allows for a

faster design and implementation process, as there is greater understanding of the model (including its limitations), reducing development risk. Where portability standards do not exist, it is desirable to develop new standards, such as with EAs. Creating many divergent “Trusted Sendmail” implementations to account for many MAC interfaces, for example, is clearly undesirable, both from the perspective of increased workload, risk associated with reduced review, and divergent (and conflicting) security properties.

3.3 Gradual Integration via the Open-source Approach

The open source development processes differ substantially from many commercial development approaches. The distributed volunteer-oriented development process reinforces a number of design and development trends, resulting in a rapid development cycle, featurism, integration of experimental features, and diverse models of “success”.

For many open source projects, the motivations for developers are different from those of closed source commercial products: they are highly motivated to do the work, but often have limited resources to bring about the results they seek. This can result in a “many testers but few developers” syndrome for features that are either less popular or technically difficult to implement. The volunteer nature of the work means that the model of success is often based on the degree to which the software is available and used, and the effectiveness in attracting new developers to a project, rather than monetary compensation.

The limitations of version control and collaboration tools often drive the organization of open source software projects that use them. For example, CVS’s inability to effectively handle a three-tier development process (central repository, per-project repository, local development tree) makes it difficult to track the rapidly moving central FreeBSD source repository without pushing changes back into the central source tree. This further encourages the wide-spread open source technique of providing early access to work

still under development, allowing for broader exposure of the code and therefore more effective testing. Providing early access to the EA implementation greatly facilitated the development of TrustedBSD features by permitting independent development of other features, otherwise made difficult by CVS’s inability to handle a hierarchical model. Likewise, allowing early access to the ACL implementation, even though it was still partially complete, allowed for far broader testing and greater numbers of developers.

Releasing early and often during the development process often means submitting the necessary hooks to support easier development, such as reserving system call numbers and adding prototype interfaces. These techniques are appropriate where hooks and interfaces are intended to remain relatively static, but allow the feature under development to generate few modification conflicts even as the base tree moves forward. For example, a number of the TrustedBSD APIs appeared in the 4.x-STABLE FreeBSD release branch, although the underlying implementations were not present. The development of improved abstractions and modular service interfaces allows the development process to be further streamlined—as better abstractions are introduced, the changes to the base source distribution necessary to support new features get progressively smaller.

The open source development process also allows a new element to be introduced in the software portability process: direct code sharing to improve interface portability. This facilitates the development of parallel implementations in a number of ways: the code may be directly “borrowed” from another distribution if the licenses are compatible, direct inspection of parallel code can improve consistency and correctness, and it is possible to take advantage of the source code for third party tools relying on the service to perform testing. The TrustedBSD project has frequently made use of open access to other systems’ source to understand the interfaces and implementation quirks of services on those systems. Implementing ACLs, for example, was greatly facilitated by the ability to recompile and test the Linux `getfacl` and `setfacl` tools on FreeBSD to determine that

they behaved consistently with the FreeBSD implementations, and that our ACL library routines behaved correctly.

For the TrustedBSD Project to succeed, it must leverage the benefits of the open source model while avoiding the pitfalls: in general, this means adapting the development cycle and processes to that of the FreeBSD Project, which has shown remarkable success in navigating the challenges of distributed collaboration and development. Understanding the social aspects of open source software development is also important, including accepting the open source success model, leveraging distributed development and testing, and using open source as a tool for improved portability.

4 Case Studies in Feature Implementation

These design and implementation goals outline a strategy for the development and integration of new advanced security features into the FreeBSD operating system. We now consider a number of the features under development as part of the TrustedBSD project, and how these goals have influenced the design and development process.

4.1 Regression tests

One of the primary challenges and risks of security feature development is that of correctness: unlike many areas of software authoring where it is possible to accurately capture the common case and then gradually address the less visited code paths, a single failure in security software can render the entire construction useless. This is especially true when introducing complex security features such as mandatory access control, where many components act in concert over a spectrum of system abstractions. As such, an important tool for successfully developing security features is a comprehensive set of tests and evaluation mechanisms, which can be used to analyze and quantify the existing implementation, then to perform incremental

verification of any changes made. Part of the challenge also lies in that FreeBSD is not a perfect starting point: the existing implementation suffered from a number of inconsistencies which had to be understood—often these existed precisely because such tools were not available.

TrustedBSD testing tools fit into two general categories: tests intended to evaluate the correctness of specific aspects of the implementation, and tests intended to evaluate the overall correctness of larger scenarios. Smaller context-specific tests attempt to exhaustively explore the behavior of a specific piece of access control or related security functionality by constructing the relevant characteristic arguments and context, then comparing the results of the function with declared expectations.

An example of this includes the `proc_to_proc` regression test, which was developed to explore the correctness of authorization policy for inter-process system calls. Inter-process calls typically involve two processes: the first (subject) process invokes a system call which will affect another (object) process. Depending on the credentials associated with processes, and the security model in use, the kernel should reject some calls, and accept others. For example, the `ptrace()` system call allows a process to attach debugging services to another process, permitting it to read and write the memory contents and state of the process, as well as control its execution flow. Such a service allow the subject process to gain access to any resources available to object process, and as such, constitutes a substantial security risk if not properly protected. The following sample output from `proc_to_proc` illustrates a test failure when a process successfully signals another process instead of receiving the `EPERM` error.

```
[21. unpriv1 on daemon1].signal: expected
      EPERM, got 0
(e:1000 r:1000 s:1000 P_SUGID:0)
(e:1000 r:0 s:0 P_SUGID:1)
```

Larger scenario tests attempt to explore whether more general expectations for cor-

rect behavior are met by the system. These tests typically perform compound operations, checking only that, given the correct starting state and sequence of operations, the desired end property is present. For example, the `setuid_protected` test evaluates whether or not a process that has executed a `setuid` binary undergoes the expected credential transformation, and, if so, is then protected from manipulation by other processes present in the system.

In both types of test, a clear notion of “correct” and an understanding of potential failure modes is required to design useful and complete tests. This is not a challenge unique to informal regression testing of security functionality on open source operating systems, but is complicated by a lack of clarity as to what the intended model should be.

The regression test design and implementation task offers substantial benefits to both TrustedBSD developers, and to the broader FreeBSD community. The test suites have already been used to simplify a number of access control checks, as well as point out inconsistencies in access control implementation. By using these tests in the development process, it is possible to gain greater assurance that the new features being added are implemented correctly, and that they do not weaken existing protections.

4.2 Extended Attributes

Extended attributes (EAs) provide a clean abstraction for associating additional metadata with files and directories, a requirement for the implementation of many new kernel security features (ACLs, Capabilities, MAC, ...) as well as a feature which non-security applications may find useful. Providing a metadata storage abstraction reduces the implementation overhead associated with these features, both in terms of redundant work and the substantial complexity of extending on-disk storage formats. Specifying a clean API allows new services to be implemented without knowledge of underlying format, permitting a rapid EA implementation early in the project to facilitate development of a number

of other features.

The EA interface provides simple semantics: for each file or directory, zero or more names may be defined. EA names exist in disjoint namespaces, of which two are defined: `EXTATTR_NAMESPACE_SYSTEM` and `EXTATTR_NAMESPACE_USER`. Namespaces determine the protection properties of an EA—access to the system namespace is limited to the kernel and privileged processes, while EAs in the user namespace are protected using the discretionary and mandatory protections on the file or directory. Each defined name may have zero or more bytes of data associated with it. No EAs are defined for a newly created file or directory, although consumers of EAs may define names and values during the creation process. Two operations are defined, allowing EAs to be atomically retrieved and set.

For a first implementation, we selected a simple design that permitted us to move on to additional new features that rely on EAs, allowing later performance optimization by those with greater expertise in file systems. Rather than modify the on-disk file system format, we chose to store EA data in backing files. This allowed us to avoid a lengthy and bug-prone development process, avoid conflicts with other on-going development on FFS, and avoid requiring low-level file system modifications to allow developers and users to experiment with EAs or features that rely on them. Each backing file stores one named EA from a single namespace for all files in the file system, and is treated as an array of EA instances indexed by inode number. Both the file itself and each instance of an EA have headers. The file header contains a backing file format version, as well as a field defining maximum size any EA instance can take on, permitting the array record size to be calculated as the sum of the EA instance header size and maximum EA instance size. EA instance headers indicate whether or not the instance is defined for the given inode, the size of the EA instance if defined, as well as a copy of the inode generation number, used for synchronization purposes. A privileged user process can invoke the `extattrctl()` system call to start EA support on a given UFS-based file system, and then enable individual EAs by

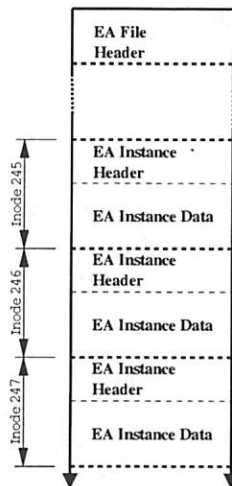


Figure 1: EA backing file format

associating backing files with EA names and namespaces.

It is also possible to have EAs automatically started and enabled for the file system at mount-time by specifying the `UFS_EXTATTR_AUTOSTART` kernel option. When enabled, the mount code will search the `.attribute/system` and `.attribute/user` directories off of the file system root for valid backing files. When a file is found, an EA with the same name is enabled in the appropriate namespace. This permits atomic starting of EA services with the mount operation, preventing race conditions that might be present as a result of a delay in EAs becoming available while other files in the file system are accessible.

This implementation offers acceptable performance, requiring an additional seek for most operations if the EA has not already been loaded from disk. Currently, the UFS EA implementation relies on the file system buffer cache to cache the backing file, rather than implementing a custom EA cache; the temporal locality properties of most services currently layered on EAs allow this caching to be effective in mitigating most performance costs.

This implementation is sufficient to implement services such as ACLs, Capabilities, and MAC above the EA interface. However, it

suffers from a number of limitations, including the treatment of EA meta-data as “data” from the perspective of the file system synchronization policy, in particular, with regards to the soft updates mechanism used in FFS. One important synchronization failure mode occurs if an EA is not always enabled when the file system is active. In this scenario, two problems arise: first, EAs are not garbage collected at file deletion, and second, services relying on EAs cannot update meta-data. The inode generation number replication into EA instance headers permits some synchronization problems to be detected, by preventing old EA data from being used with a new file, as the inode generation number is changed when the inode is re-allocated. In large part, the service meta-data update problem is solved by allowing the atomic auto-starting of EAs at mount-time. Currently, work is in the planning stages for a block-level implementation in FFS, which would have stronger performance and consistency properties while retaining the same interface, requiring no change to services above it.

4.3 Access Control Lists

Access Control Lists (ACLs) allow users to express more detailed policies for files and directories that they own. POSIX.1e defines an ACL interface that acts as a superset to the current permission mechanism: the file access ACL consists of a base ACL derived from the file permissions, and the extended ACL defines permissions for additional users, groups, and an optional ACL mask. Each non-mask entry in the ACL associates a user or group with a set of rights that the user or group will have on the file or directory.

The ACL evaluation algorithm selects an appropriate part of the credential and an entry in the ACL that are combined during permission evaluation; this order of preference matches first the owner, then additional user entries, then group entries, and finally, the “other” entry. The POSIX.1e ACL mask plays an important role in providing compatibility for ACL-unaware programs: it places a bound on the maximum rights provided

by any additional users or group entries. If an extended ACL is available for an inode, the `chmod()` operation on the file is modified: rather than setting the file group bits, the ACL mask is modified. As a result, modification of the group bits in the permission effectively masks the rights for all entries of the ACL other than the file owner and other entries, allowing programs not aware of ACL interfaces to place an upper bound on file accessibility. Additional compatibility is provided by a default ACL placed on directories, which is combined with the permission set provided by the process on `open()` or `create()` to produce the new access ACL for a file created in that directory, allowing ACL-unaware applications to create a new file with an appropriate ACL.

The FreeBSD implementation splits the ACL data over the existing inode mode field in UFS, and two EAs, `posix1e.acl_access` for the access ACL on an inode, and `posix1e.acl_default` for the default ACL. At the VFS layer, two new vnode operations are introduced: `VOP_GETACL` to retrieve available ACLs from a vnode, and `VOP_SETACL` to set ACLs on the vnode. The caller may specify the ACL type determining whether the ACL operation is intended for the access or default ACL. When ACL support is compiled into the kernel, ACL code is enabled in a number of other UFS vnode operations, including `VOP_ACCESS` which invokes a generic `vaccess_acl_posix1e()` access check routine, as well as during file and directory creation via `VOP_CREATE()`, `VOP_MKNOD()`, `VOP_MKDIR()`, and `VOP_SYMLINK()`, where the default ACL, if any, is combined with the requested file mode to produce the access ACL for the child. As the ACL is split over both the inode mode and EA storage, the fields must be synchronized during certain operations—in particular, the ACL vnode operations, but also during file creation to combine the default ACL and request mode.

As a result of splitting the access ACL in this manner, many frequently performed operations, such as `stat()` and `chmod()` incur no additional overhead. The access ACL must be read for `open()` and `access()` calls on a file, and during actual ACL read or update operations. Access ACLs impose a

slightly higher cost on directory operations than on file operations, although they also exhibit higher locality: directory lookup and listing requires that the access ACL be evaluated for the `ACL_EXECUTE` and `ACL_READ` permissions, respectively. Creation of a new file or sub-directory within a directory also exhibits higher cost because both the access and default ACLs must be retrieved for the parent, and then new access and default ACLs may be written out for the child.

In practice, ACL operations have high temporal locality lending them to caching, and suffer from higher latency rather than actual disk I/O utilization increase. When ACLs are not enabled on the file system, there is no measurable performance difference from the pre-ACL implementation, in keeping with the “minimal impact on current configurations” mandate. When ACLs are enabled but not used, an overhead is perceived due to reads associated with determining if an access ACL is defined, and for the lookup of default ACLs during file or sub-directory creation. When ACLs are enabled and utilized, higher costs are perceived during file and sub-directory creation if a default ACL is set on the directory in which new children are created. To improve the actual cost of ACLs when in use, the primary target for optimization is the EA implementation: the measured costs of ACL operations is effectively identical to the measured cost of the EA operation supporting the ACL operation.

The POSIX.1e ACL specification offers a largely complete and unambiguous specification for an ACL implementation; some extensions, however, are required to add more complete functionality in FreeBSD, such as the ability to perform ACL operations on directories via a file handle. Although the ACL mask behavior increases complexity, it provides relatively transparent support for ACL-unaware applications. While the ACL specification is not identical to the variations used in many commercial UNIX variants, it offers compatible semantics. The ACL implementation will be included in FreeBSD 5.0-RELEASE, and a number of applications, including Samba, already work properly with ACLs on FreeBSD 5.0-CURRENT development branch.

4.4 Mandatory Access Control (MAC)

Mandatory access control permits security administrators to specify fine-grained policies limiting the interactions between users and objects on the system, which will be enforced regardless of the any permissions granted by discretionary access control primitives. Often, mandatory access control policies consist of schemas for limiting information flow, such as MLS and Biba policies, but may also consist of more general policies, such as Type Enforcement, or more specific policies, such as the FreeBSD jail mechanism. MAC policies generally require that subjects and objects be labeled with the necessary administrative information to support the policy, which might include information sensitivity or integrity levels, an assigned data type, or the index or name of a compartment. They also require a broad set of enforcement points across a majority of operating system operations.

An initial experimental implementation has provided the desired functionality of enforcing three fixed MAC policies: MLS, a fixed-label Biba policy, and a generalization of the native FreeBSD Jail compartmentalization policy. In the long term, we hope to provide a more general framework for introducing mandatory access control mechanism. The policies are enforced over a fairly wide set of system objects, including processes as the target for inter-process operations, system management objects such as `sysctl` nodes, file system objects such as files and directories, and network objects such as sockets, interfaces, and `mbufs`. MAC labels are described by a `struct mac` which is appropriate for use on both subjects and objects, and currently contains three fields relevant to the three policies.

To support the labeling of subjects (processes), the `ucred` structure is extended to include an additional `struct mac`. `cred0`, the process credential for the first kernel process, is initialized to high integrity, low secrecy, and is not present in any jail compartment. All other processes inherit this credential, unless an intermediate process has modified it; priv-

ileged processes are permitted to update the MAC fields in accordance with the MAC policies. The user login mechanisms have been updated to retrieve per-user label information from the `login.conf` user class data. This requires that components of the system making use of the `setusercontext()` call now also set the `SET_MACLABEL` flag. Eventually, additional sources of information, such as incoming terminal and network label, may be used to make a policy-driven label determination.

For inter-process authorization, the existing `p_can*()` primitives were modified to call the `mac_uca*()` versions of the call which could return a new failure mode.

To handle the labeling of transient kernel objects, a new label structure was created, `struct objlabel`, which contains the necessary ownership and protection information, including owner and ACL, as well as a `struct mac` for mandatory protection. `struct objlabel` behaves in a similar manner to `struct ucred`, in that a set of initial object labels are initialized by appropriate kernel subsystems, and then inherited (copy-on-write) by various children objects. For example, packets inherit the object label of the interface they originate from. For objects created by subjects, the new object label is based on a composition of the subject credential, and possible object parents. A series of new access control check primitives were introduced that check authorization between subject credentials and object labels, and were liberally scattered through system operations.

Some objects, such as sockets, play the interesting role of both subject and object: FreeBSD caches the subject's credential with the socket on creation, which allows the properties of the socket to remain static when transferred or inherited; this also allows UID-based decisions to be made on delivery of packets to sockets in the `ipfw` firewall code. This permits MAC delivery decisions to be made at the network layer without directly inspecting the receiving process or dealing with the ambiguity of multiple processes having access to a single socket. However, sockets are also objects when written to or read from by processes that have access to them, and

therefore have an object label. Both types of events (acting on the socket as a subject and as an object) require mediation.

Currently, file system objects do not make use of the object label abstraction, instead mapping MAC labels into EAs on the file system, reading them when an access control check must be made. A new access control primitive, `vaccess_mac()` accepts subject credentials, vnode properties, and MAC labels loaded from EAs, and returns an access control decision which is then composed with the results of the discretionary access control check, `vaccess_acl_posix1e()` to generate a final access control result. In the future, we will look at allowing file systems to maintain `objlabel` structures directly, improving their ability to utilize more general abstractions.

Many MAC implementations make use of poly-instantiation to resolve namespace use conflicts by processes with conflicting labels. For example, UNIX processes may expect to be able to write files to `/tmp` at will—however, information flow policies may not permit a process with one integrity level to be aware of files written to the directory by a process with a lower integrity level. If the two processes select the same file name, under traditional UNIX semantics, one process will receive an error: this is not permitted under information flow MAC policies. Poly-instantiation allows different processes to appear to address the same namespace while being partitioned from one another: in the case of the file system, this might mean that the `namei()` name lookup routine points the processes at different underlying directories. TrustedBSD does not currently implement automatic poly-instantiation for directories, or for other namespaces such as the IP port and System V IPC namespaces, and in that sense, is incomplete. For the purposes of processes making use of the `/tmp` directory, appropriate setting of the `TMPDIR` environment variable has proven sufficient for the present—however, in the future, this issue will need to be addressed.

Since this is still a highly experimental environment, performance figures are not yet available, but appear to be similar to those of ACLs: when not involving file system ac-

cesses, the performance cost for most objects is negligible; when an EA operation is required, the performance corresponds to the required EA operations. The impact on the network subsystem is of particular interest, as new label operations are now interposed on existing packet and interface operations, and may impose a performance hit. Future MAC work on FreeBSD will include improved abstractions for managing labels, more pervasive use of these abstractions, such as in the file systems, and implementation of features such as poly-instantiation, processes making use of ranges of labels to mediate access between normally isolated process classes, and work to measure and optimize performance.

5 Lessons Learned

A number of important lessons relating to both open source and security development can be derived from the experience of introducing the current TrustedBSD feature set into the FreeBSD operating system, and will be carried into future work.

- *Adapt to an ill-defined starting point.* While FreeBSD is characterized by strong architectural design, security functionality has never been a specific target. As a result, substantial cleanup was required to bring the starting point in line with expectations. This is actually a benefit, as it provides the opportunity to educate the broader community about security requirements, and to grow a more complete understanding of current use.
- *Incremental improvement can result in good software.* While incremental development can introduce security problems, as seen with inconsistent access control checks, introducing interfaces and features incrementally during the overall software development process of a larger project reduces the workload for developers by reducing the cost of testing and source code merging. That is to say, “release early and often” can be rephrased as “release early and merge often” for

projects that build off an existing project that constitute a moving target.

- *Improved abstractions for existing features support most new features.* Current security functionality in FreeBSD provided rationale for the abstraction improvements necessary for new security work. Making these abstraction improvements has had a high payoff in terms of improving flexibility to introduce new security models and services.
- *Being part of the developer community provides credibility to make more far-reaching changes from within, rather than "throwing the changes over the fence".* To have greatest impact, it is necessary to be part of the development community, working with that community to reach design decisions that are acceptable to all relevant parties.
- *Cross-platform portability efforts have high pay-off.* Working to build cross-platform consensus on security interfaces can have a high payoff: this has already proven the case with Samba support for POSIX.1e ACLs, and is likely to continue to be the case moving forward with security integration into other common applications.
- *Time invested in public relations activity is time well spent.* The open source community is driven by a desire for features and improvement, and revolves around a relatively small set of online forums. Investing time in publicizing work and building credibility can have very positive results in terms of generating broad support for the features, and may be vital when it comes to portability work.

6 Future Directions

TrustedBSD remains very much a work in progress: while a number of features and interfaces have been successfully integrated, much of the work remains to be done. There are several areas in which particular attention will be focused:

- *Extended attribute interface.* Substantial progress has been made in defining and implementing a general meta-data service for UNIX-style file systems. However, substantial portability work remains to be done so that applications can be assured consistent interfaces for accessing meta-data on FreeBSD and other platforms. For the TrustedBSD project, this will involve porting the EA mechanism to other operating systems, including the currently targeted OpenBSD and Darwin platforms; it will also involve continued discussion with members of the Linux community.
- *Mandatory access control.* With one and a half experimental MAC implementations under the bridge, the experience gained is becoming sufficient to look at integrating some of the components of the MAC implementations back into the base system. In particular, this involves abstraction improvements such as generalized object labeling, which permit the association of security labels with arbitrary kernel objects. MAC is an important feature to many potential TrustedBSD consumers, and therefore represents the next major integration challenge.
- *Audit.* An important trusted operating system feature unmentioned in this paper is that of event auditing. While implementing event auditing has been a goal of the project since it started, audit will most likely represent a serious challenge. In part, this is because auditing support requires widespread and intrusive changes throughout the kernel to gather information, as well as posing a substantial performance burden.
- *Plugability.* A long-term goal of the project is to improve the modularity of security services within the kernel so that they may be easily extended or replaced. This is possible through improved abstractions, and many examples of extensible kernel subsystems exist on which to base this work, including the Virtual File System (VFS)[21].
- *Documentation.* As the availability of TrustedBSD features increases for the

broader FreeBSD community, documentation and education will play an increasing role in the project's work, to keep both system developers and users abreast of the new services.

7 Related Work

There is a long history of research and development relating to trusted operating systems. In the area of access control, there has been extensive research into various types of discretionary and mandatory control models[3][4][6], evolution of these models into standards and requirements[16] [9][5][11][10], and improved abstractions and models derived from these experience[2][13].

A number of trusted systems have been developed in the form of both research operating systems, and extensions to existing commercial systems. Trusted Mach[19] and other experimental trusted operating systems have explored the impact of secure design when building from the ground up. Many UNIX vendors offer trusted versions of their systems built in-house, such as SGI's Trusted IRIX[18]. There are also operating system security extension products that introduce trusted operating system features, such as PitBull from Argus Systems[1]. Open source trusted system work includes the LOMAC extensions for Linux[6], SELinux[13], POSIX.1e ACLs[8] and Capabilities[15] for Linux, and the Linux RSBAC project[17].

The TrustedBSD project benefits from both past and current research, building on the exploration of access control requirements and models, as well as research into improved abstractions and interfaces.

8 Conclusion

TrustedBSD provides a set of trusted operating system extensions to the FreeBSD operating systems. Through close cooperation with the FreeBSD community, a tight integra-

tion between the security features and base services of the operating system will be possible. The challenges in such an environment are both technical and social, where tasks from both categories play an important role in the success of the project. Not least of the challenges is the education of FreeBSD developers and users regarding new features.

9 Acknowledgments

Large parts of this work were done in co-operation with the FreeBSD and TrustedBSD development communities. In particular, I thanks Chris Faulhaber, Thomas Moestl, Ilmar Habibulin, and Brian Feldman for their participation in developing and debugging these features. In addition, my thanks to Ruslan Ermilov, Dima Dorfman, and Chris Costello for documentation support.

A major focus of the TrustedBSD work has been to emphasize portability of the feature sets, particularly with other open source operating systems. Both Andreas Gruenbacher, author of the Linux ACL and EA implementations, and Andrew Morgan, author of the Linux Privileges implementation, have been vital to this approach through their discussion of the POSIX.1e specification, and implementation feedback and critique. Thanks also to the Trust Technology group at SGI, including Casey Schaufler, Richard Offer, and Linda Walsh, all of whom have provided feedback on the POSIX.1e specification, and system/application requirements.

Substantial contributions of funding, development resources, and travel and communication reimbursement have been provided by NAI Labs, BSDi, Safeport Network Services, without whom the TrustedBSD Project would not have been possible.

10 Availability

The TrustedBSD Extensions to FreeBSD are distributed under a two-clause Berkeley-

style license, encouraging integration into both open and closed source products. During development, patches and code are available via the TrustedBSD web site: <http://www.TrustedBSD.org/>

As features reach maturity, they are integrated back into the base FreeBSD distribution: <http://www.FreeBSD.org/>

References

- [1] Argus products overview: Pitbull. <http://www.argussystems.com/product/overview/pitbull/>.
- [2] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A domain and type enforcement UNIX prototype. In *Computing Systems, Winter, 1996.*, volume 9, Berkeley, CA, USA, Winter 1996. USENIX.
- [3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.
- [4] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA, Apr. 1977.
- [5] N. C. C. I. Board. Common criteria version 2.1 (ISO IS 15408), 2000.
- [6] T. Fraser. LOMAC: MAC You Can Live With. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.
- [7] FreeBSD Project. FreeBSD home page. <http://www.FreeBSD.org/>.
- [8] A. Grunbacher. Extended attributes and access control lists for linux. <http://acl.betbits.at/>.
- [9] IEEE. Portable operating system interface (POSIX)—part 1: System application program interface (API): Protection, audit and control interfaces: C language, October 1997. PSSG/D17, POSIX.1e.
- [10] N. S. A. Information Systems Security Organization. Controlled access protection profile version 1.d, October 1999.
- [11] N. S. A. Information Systems Security Organization. Labeled security protection profile version 1.b, October 1999.
- [12] P.-H. Kemp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings, SANE 2000 Conference*. NLUUG, 2000.
- [13] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.
- [14] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. Design and implementation of the 4.4BSD operating system, 1996.
- [15] A. Morgan. Privileges for linux. <http://www.kernel.org/pub/linux/libs/security/linux-privs/>.
- [16] U. S. D. of Defense. *Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985.
- [17] Rule set based access control (RSBAC) for linux. <http://www.rsbac.org/>.
- [18] SGI. B1 sample source code. <http://oss.sgi.com/projects/ob1/>.
- [19] Trusted Mach Security Architecture. TIS TMACH Edoc-0001-97A.
- [20] TrustedBSD Project. TrustedBSD home page. <http://www.TrustedBSD.org/>.
- [21] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In *Proceedings: USENIX Association Winter Conference, January 23-25, 1985, Dallas, Texas, USA*, pages 117-124. USENIX, Winter 1985.
- [22] R. Watson. Introducing supporting infrastructure for trusted operating system support in FreeBSD. In *BSD Conference*, Monterey, CA, USA, October 2000.

Integrating Flexible Support for Security Policies into the Linux Operating System

Peter Loscocco, NSA, loscocco@tycho.nsa.gov
Stephen Smalley, NAI Labs, ssmalley@nai.com

Abstract

The protection mechanisms of current mainstream operating systems are inadequate to support confidentiality and integrity requirements for end systems. Mandatory access control (MAC) is needed to address such requirements, but the limitations of traditional MAC have inhibited its adoption into mainstream operating systems. The National Security Agency (NSA) worked with Secure Computing Corporation (SCC) to develop a flexible MAC architecture called Flask to overcome the limitations of traditional MAC. The NSA has implemented this architecture in the Linux operating system, producing a Security-Enhanced Linux (SELinux) prototype, to make the technology available to a wider community and to enable further research into secure operating systems. NAI Labs has developed an example security policy configuration to demonstrate the benefits of the architecture and to provide a foundation for others to use. This paper describes the security architecture, security mechanisms, application programming interface, security policy configuration, and performance of SELinux.

1 Introduction

End systems must be able to enforce the separation of information based on confidentiality and integrity requirements to provide system security. Operating system security mechanisms are the foundation for ensuring such separation. Unfortunately, existing mainstream operating systems lack the critical security feature required for enforcing separation: mandatory access control (MAC) [17]. Instead, they rely on discretionary access control (DAC) mechanisms. As a consequence, application security mechanisms are vulnerable to tampering and bypass, and malicious or flawed applications can easily cause failures in system security.

DAC mechanisms are fundamentally inadequate for strong system security. DAC access decisions are only based on user identity and ownership, ignoring other security-relevant information such as the role of the user, the function and trustworthiness of the program, and the sensitivity and integrity of the data. Each user has complete discretion over his objects, making it impossible to

enforce a system-wide security policy. Furthermore, every program run by a user inherits all of the permissions granted to the user and is free to change access to the user's objects, so no protection is provided against malicious software. Typically, only two major categories of users are supported by DAC mechanisms, completely trusted administrators and completely untrusted ordinary users. Many system services and privileged programs must run with coarse-grained privileges that far exceed their requirements, so that a flaw in any one of these programs can be exploited to obtain complete system access.

By adding MAC mechanisms to the operating system, these vulnerabilities can be addressed. MAC access decisions are based on labels that can contain a variety of security-relevant information. A MAC policy is defined by a system security policy administrator and enforced over all subjects (processes) and objects (e.g. files, sockets, network interfaces) in the system. MAC can support a wide variety of categories of users on a system, and it can confine the damage that can be caused by flawed or malicious software.

Traditional MAC mechanisms have typically been tightly coupled to a multi-level security (MLS) [7] policy which bases its access decisions on clearances for subjects and classifications for objects. This traditional approach is too limiting to meet many security requirements [8, 9, 10]. It provides poor support for data and application integrity, separation of duty, and least privilege requirements. It requires special trusted subjects that act outside of the access control model. It fails to tightly control the relationship between a subject and the code it executes. This limits the ability of the system to offer protection based on the function and trustworthiness of the code, to correctly manage permissions required for execution, and to minimize the likelihood of malicious code execution.

To address the limitations of traditional MAC, the National Security Agency (NSA), with the help of Secure Computing Corporation (SCC), began researching new ways to provide strong mandatory access controls that could be acceptable for mainstream operating systems. An important design goal for the NSA was to provide

flexible support for security policies, since no single MAC policy model is likely to satisfy everyone's security requirements. This goal was achieved by cleanly separating the security policy logic from the enforcement mechanism. Through the development of two Mach-based prototypes, DTMach [12] and DTOS [20], the NSA and SCC developed a strong, flexible security architecture. Although high assurance was not a goal of the research, formal methods were applied to the design to help validate the security properties of the architecture [23, 24]. Likewise, performance optimization was not a goal, but significant steps were taken in the architecture to minimize the performance overhead normally associated with MAC. NSA and SCC then worked with the University of Utah's Flux research group to transfer the architecture to the Fluke research operating system [25]. During the transfer, what has become the *Flask* architecture was enhanced to provide better support for dynamic security policies.

The NSA created *Security-Enhanced Linux*, or *SELinux* for short, by integrating this enhanced architecture into the Linux operating system. It has been applied to the major subsystems of the Linux kernel, including the integration of mandatory access controls for operations on processes, files, and sockets. NAI Labs has since joined the effort and has implemented several additional kernel mandatory access controls, including controls for the *procfs* and *devpts* file systems. The MITRE Corporation and SCC have contributed to the development of some application security policies and have modified utility programs, but their contributions are not discussed further in this paper.

Using the flexibility of SELinux, it is possible to configure the system to support a wide variety of security policies. The system can support:

- separation policies that can enforce legal restrictions on data, establish well-defined user roles, or restrict access to classified data,
- containment policies useful for such things as restricting web server access to only authorized data and minimizing damage caused by viruses and other malicious code,
- integrity policies that are capable of protecting unauthorized modifications to data and applications, and
- invocation policies that can guarantee data is processed as required.

The flexibility of SELinux meets the goal of enabling many different models of security to be enforced with the same base system.

The NSA released the SELinux to make the technology available to a wider community and enable further research into secure operating systems. To help introduce the system in a more immediately useful form that helps demonstrate the added value of SELinux, NSA contracted NAI Labs to develop an example security policy configuration for the system designed to meet a number of common general-purpose security objectives. The example configuration greatly reduces the complexity of SELinux that would otherwise be present if building the policy specification from scratch were required. The example configuration released with the SELinux provides a customizable foundation with which a secure system can be built.

The remainder of this paper describes SELinux. It begins by providing an overview of the Flask architecture and its SELinux implementation in Section 2. The security mechanisms added to the system are then described in Section 3. The SELinux application programming interface (API) is discussed in Section 4. Section 5 describes the example security policy configuration created for the system. The performance overhead of the SELinux mechanisms is described in Section 6. Related work is discussed in Section 7.

2 Security Architecture

This section provides an overview of the Flask architecture and the SELinux implementation of the architecture. The Flask architecture provides flexible support for mandatory access control policies. In a system with mandatory access controls, a security label is assigned to each subject and object. All accesses from a subject to an object or between two subjects must be authorized by the policy based on these labels. The Flask architecture cleanly separates the definition of the policy logic from the enforcement mechanism. The security policy logic is encapsulated within a separate component of the operating system with well-defined interfaces for obtaining security policy decisions. This separate component is referred to as the *security server* due to its origins as a user-space server running on a microkernel. In the SELinux implementation, the security server is merely a kernel subsystem.

Components in the system that enforce the security policy are referred to as *object managers* in the Flask architecture. Object managers are modified to obtain security policy decisions from the security server and to apply these decisions to label and control access to their objects. In the SELinux implementation, the other kernel subsystems (e.g. process management, filesystem, socket IPC, System V IPC) are object managers. Application object managers can also be supported, such as a windowing system or a database management system.

The Flask architecture also provides an access vector cache (AVC) component that stores the access decision computations provided by the security server for subsequent use by the object managers. The AVC component also supports revocation of permissions, as described later in Section 2.4. An object manager may further reduce the cost of a permission check by storing references to the appropriate entry in the AVC with its objects. As a result, most permission checks can occur without even incurring the cost of an extra function call.

The remainder of this section further elaborates on the Flask architecture and its SELinux implementation. It begins by discussing how security labels are encapsulated in Flask. This section then discusses how Flask supports flexibility in labeling and access decisions. The ability of Flask to support policy changes is then described.

2.1 Encapsulation of Security Labels

Since the content and format of security labels are dependent on the particular security policy, the Flask architecture defines two policy-independent data types for security labels: the security context and the security identifier. A security context is a variable-length string representation of the security label. Internally, the security server stores a security context as a structure using a private data type. A security identifier (SID or *security_id_t*) is an integer that is mapped by the security server to a security context. Flask object managers are responsible for binding security labels to their objects, so they bind SIDs to active kernel objects. The file system object manager must also maintain a persistent binding between files and security contexts. Since the object managers handle SIDs and security contexts opaquely, a change in the format or content of security labels does not require any changes to the object managers.

The Flask architecture merely specifies the interfaces provided by the security server to the object managers. The implementation of the security server, including any policy language it may support, are not specified by the architecture. The SELinux example security server defines a security policy that is a combination of Type Enforcement (TE) [8], role-based access control (RBAC) [11], and optionally multi-level security (MLS) [7]. The example configuration for the TE and RBAC policy components is described in Section 5. The SELinux example security server defines a security context as containing a user identity, a role, a type, and optionally a MLS level or range. Roles are only relevant for processes, so file security contexts have a generic *object_r* role. The security server only provides SIDs for security contexts with legal combinations of user, role, type, and level or range. The individual attributes of the

```
int security_transition_sid(
    security_id_t ssid,
    security_id_t tsid,
    security_class_t tclass,
    security_id_t *out_sid);

ret = security_transition_sid(
    current->sid,
    dir->i_sid,
    SECCCLASS_FILE,
    &sid);
```

Figure 1: Interface and example call to obtain a security label. The input parameters are the subject SID, the SID of a related object (e.g. the parent directory), and the class of the new object. The SID for the new object is returned as an output parameter.

security context are not manipulated by the object managers.

The user identity attribute in the security context is independent of the ordinary Linux user identity attributes. Modifications to the Linux login program and cron daemon are provided to set this new user identity attribute appropriately for login sessions and user cron jobs. By using a separate user identity attribute, the SELinux mandatory access controls remain completely orthogonal to the existing Linux access controls. SELinux can enforce rigorous controls over changes in its user identity attribute without affecting compatibility with existing applications.

2.2 Flexibility in Labeling Decisions

When a Flask object manager requires a label for a new object, it consults the security server to obtain a labeling decision based on the label of the creating subject, the label of a related object, and the class of the new object. For program execution, the Flask process manager obtains the label for the transformed process based on the current label of the process and the label of the program executable. For file creation, the Flask file system object manager obtains the label for the new file based on the label of the creating process, the label of the parent directory, and the kind of file being created. The security server may compute the new label based on these inputs and may also use other external information. Figure 1 shows the security server's *security_transition_sid* interface for obtaining a label and an example call to this interface to obtain the label of a new file.

The SELinux example security server may be configured to automatically cause changes in the role or domain attributes of a process based on the role and domain of the process and the type of the program. By default, the role and domain of a process is not changed by program execution. The SELinux security server may also be configured to use specified types for new files based

```
int security_compute_av(
    security_id_t ssid,
    security_id_t tsid,
    security_class_t tclass,
    access_vector_t requested,
    access_vector_t *allowed,
    access_vector_t *decided,
    __u32 *seqno);
```

Figure 2: Interface for obtaining access decisions from the security server. The input parameters are a pair of SIDs, the class of the object, and the set of requested permissions. The pair of SIDs may be subject-to-object, subject-to-subject, or even object-to-object. The granted permissions are returned as output parameters.

on the domain of the process, the type of the parent directory, and the kind of file. A new file inherits the same type as its parent directory by default. For objects where there is only one relevant SID, object managers typically do not consult the security server. Instead, they merely use this SID as the SID for the new object. Pipes, file descriptions, and sockets inherit the SID of the creating process, and output messages inherit the SID of the sending socket.

2.3 Flexibility in Access Decisions

Object managers consult the AVC to check permissions based on a pair of labels and an object class, and the AVC obtains access decisions from the security server as needed. Figure 2 shows the security server's *security_compute_av* interface for obtaining access decisions. Figure 3 shows the AVC's *avc_has_perm_ref* interface for checking permissions and an example call to this interface to check bind permission to a socket.

Each object class has a set of associated permissions. These permission sets are represented by a bitmap called an *access vector* (*access_vector_t*). Flask defines a distinct permission for each service, and when a service accesses multiple objects, Flask defines a separate permission to control access to each object. For example, when a file is unlinked, Flask checks *remove_name* permission to the directory and *unlink* permission to the file.

The use of object classes in access requests allows distinct permission sets to be defined for each kind of object based on the particular services that are supported by the object. It also allows the security policy to make distinctions based on the kind of object, so that access to a device special file can be distinguished from access to a regular file and access to a raw IP socket can be distinguished from access to a UDP or TCP socket.

2.4 Support for Policy Changes

The Flask AVC provides an interface to the security server for managing the cache as needed for policy changes. Sequence numbers are used to address the po-

```
extern inline
    int avc_has_perm_ref(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t requested,
        avc_entry_ref_t *aeref);

ret = avc_has_perm_ref(
    current->sid,
    sk->sid, sk->sclass,
    SOCKET_BIND,
    &sk->avcr);
```

Figure 3: AVC interface and example call to check permissions. The input parameters are the same as for *security_compute_av*, except for the additional *aeref* parameter. On its first use, the *aeref* parameter is set to refer to the AVC entry used for the permission check, and on subsequent checks this reference is used to optimize the lookup. The reference is revalidated on each use to ensure its correctness.

tential interleaving of access decision computations and policy change notifications. When the AVC receives a policy change notification, it updates its own state and then invokes callback functions registered by the object managers to update any permissions retained in the state of the object managers. For example, permissions may be retained in the access rights in page tables or in the flags on an open file description. After updating the state of the object managers and the state of the AVC to conform to the policy change, the AVC notifies the security server that the transition to the new policy has been completed.

In SELinux, many permissions are revalidated on use, such as permissions for reading and writing files and permissions for communicating on an established connection. Consequently, policy changes for these permissions are automatically recognized and enforced without the need for object manager callbacks. Permissions can be efficiently revalidated by object managers using references to entries in the AVC. However, the revalidation of permissions on use is not adequate for revoking access to mapped file pages in the Linux page cache. The current SELinux implementation does invalidate the appropriate page cache entries when a file is relabeled, but a callback has not yet been defined to invalidate the appropriate page cache entries when a policy change notification is received.

The SELinux example security server provides an interface for changing the security policy configuration at runtime. The *security_load_policy* call may be used to read a new policy configuration from a file. After loading the new policy configuration, the security server updates its SID mapping, invalidating any SIDs that are no longer authorized, and resets the AVC. Subsequent permission checks on processes and objects with invalid

PERMISSION(S)	DESCRIPTION
execute	Execute
transition	Change label
entrypoint	Enter via program
sigkill sigstop sigchld signal	Signal
fork	Fork
ptrace	Trace
getsched	Get schedule info
setsched	Set schedule info
getsession	Get session
getpgid	Get process group
setpgid	Set process group
getcap	Get capabilities
setcap	Set capabilities

Table 1: Permissions for the process object class.

SIDs always fail, preventing any further accesses by such processes and any further accesses to such objects. Support for automatically relabeling these processes and objects to a label that is accessible to administrators has not yet been implemented.

3 Security Mechanisms

This section describes the security mechanisms defined by the Flask architecture and the SELinux implementation of these mechanisms. It begins with a discussion of the mandatory access controls for process management. Mandatory access controls for file system objects are then described. This section concludes with a discussion of socket mandatory access controls.

3.1 Process Controls

Table 1 shows the permissions defined for the process management component. The process *execute* permission is used to control the ability of a process to execute from a given executable image. This permission is checked between the label of the transformed process and the label of the executable on every program execution. It is also checked when an ELF or script interpreter is executed, and when a file is memory-mapped with execute access (i.e. a shared library). This process *execute* permission is distinct from the file *execute* permission which is used to control the ability of a process to initiate the execution of a program.

The *transition* permission is used to control the ability of a process to transition from one security identifier (SID) to another. The *entrypoint* permission is used to control what programs may be used as the entry point for a given process SID. This permission is similar to the process *execute* permission, except that it is only checked when a process transitions to a new SID. Hence,

the security policy can distinguish between what programs may be used to initially enter a given process SID and the full set of programs that may be executed by that process SID.

This *entrypoint* permission is especially necessary in an environment with shared libraries, since most processes must be authorized to execute the system dynamic loader. Without separate control over entry point programs, any security label could be entered by executing the system dynamic loader. Separate entry point control is also necessary in order to support security label transitions on scripts, since the new security label must be authorized to execute the interpreter and the script.

Separate permissions for each signal could easily be defined, but until empirical evidence suggests this is necessary, this will not be done. Separate permissions were defined for the *SIGKILL* and *SIGSTOP* signals, *sigkill*, *sigstop* respectively, since these signals cannot be caught or ignored. A separate permission, *sigchld* was also defined to control the *SIGCHLD* signal because experience demonstrated that it was useful to control this signal separately. A single permission, *signal*, is used to control the remaining signals.

The *ptrace* permission is used to control the ability of a process to trace another process. The *getsched*, *setsched*, *getsession*, *getpgid*, *setpgid*, *getcap*, and *setcap* permissions are used to control the ability of a process to observe or modify the corresponding attributes of another process.

In addition to the permissions listed in this table, SELinux provides an equivalent permission for each Linux capability. This allows the security policy to control the use of capabilities based on the SID of the process. SELinux could also be extended to provide a finer-grained replacement mechanism for capabilities. Such a mechanism was developed for one of SELinux's predecessors, the DTOS system [20]. This mechanism permitted privileges to be granted based on both the attributes of the process and the attributes of the relevant object, e.g. discretionary read override could be granted to a particular set of files. Since the mechanism obtained privilege decisions from the security server, management of privileges was centralized and verification that privileges were granted appropriately was straightforward.

3.2 File Controls

Table 2 lists the permissions for controlling access to open file description objects. Since open file descriptions may be inherited across *execve* or transferred through UNIX socket IPC, SELinux labels and controls open file descriptions. An open file description is labeled with the SID of its creating process, since its state is usu-

PERMISSION(S)	DESCRIPTION
create	Create
getattr	Get attributes
setattr	Set attributes
inherit	Inherit across execve
receive	Receive via IPC

Table 2: Permissions for the open file description object class.

ally treated as part of the private state of the process. It is important to distinguish between the label of an open file description and the label of the file it references. A read operation on a file changes the file offset in the open file description, so it may be necessary to prevent a process from reading a file using an open file description received or inherited from another process even though the process is allowed to directly open and read the file.

Permissions for controlling access to file systems are shown in Table 3. SELinux labels file systems and controls services that manipulate file systems, including calls for mounting and unmounting file systems, the *statfs* call and the file creation calls. SELinux controls the mounting of file systems through several permission checks. It requires that the process have *mounton* permission to the mount point directory and *mount* permission to the file system. It also requires that the *mountas-associate* permission be granted between the root directory of the file system and the mount point directory.

SELinux binds security labels to files and directories and controls access to them. It stores a persistent labeling table in each file system that specifies the security label for each file and directory in that file system. For efficient storage, SELinux assigns an integer value referred to as a *persistent SID* (PSID) to each security label used by an object in a file system. The persistent labeling table is partitioned into a mapping between each PSID and its security label and a mapping between each object and its PSID. Since the table is stored in each file system, file labels are preserved if the file system is mounted at a different location or if the file system is moved to a different system.

The mapping between each PSID and its security label is implemented using regular files in a fixed subdirectory of the root directory of each file system. This mapping is loaded into memory when the file system is mounted, and is updated both in memory and on the disk when a new security label is used for an object in the file system. The mapping between each object and its PSID is implemented by storing the PSID in an unused field of the on-disk inode. Since the PSID is available in the on-disk inode, no extra overhead is incurred either to obtain the PSID when a file is accessed or to set the PSID when a file is created. Additionally, since the mapping be-

PERMISSION(S)	DESCRIPTION
mount	Mount
remount	Change options
unmount	Unmount
getattr	Get attributes
relabelfrom relabelto transition	Relabel
associate	Associate file

Table 3: Permissions for the file system object class.

tween each object and its PSID is inode-based, changes to the file system name space do not affect the mapping.

SELinux currently only implements file labeling for the *ext2* file system. However, only the binding between on-disk inodes and PSIDs is filesystem-specific, so support for other local file system types can be easily added. For NFS file systems, a single label is currently used for all files mounted from a given NFS server. A design has been developed to provide complete file labeling and controls for NFS filesystems, but this design has not yet been implemented. SELinux also implements file labeling for the special *procfs* and *devpts* file systems based on the labels of the associated process, but these special file system types do not require the use of persistent label mappings.

When an unlabeled file system is first mounted, a persistent labeling table is created for the file system, using a default label for all files obtained from the security server. Subsequently, existing files may be relabeled using new system calls. A program called *setfiles* is used to initially set file labels from a configuration file that specifies labels based on pathname regular expressions. This program and configuration file may also be used to reset file labels to a well-defined state. However, unless the configuration file is updated to reflect runtime changes in file labels, these changes will be lost when the program is executed. Runtime changes may occur as a result of new files being created, existing files being relabeled, or changes to the name space.

Table 4 shows the permissions defined for controlling access to files, and Table 5 shows the additional permissions defined for controlling access to directories. SELinux defines a separate permission for each file and directory service. For example, SELinux defines an *append* permission for files in addition to the *write* permission, and it defines separate *add_name* and *remove_name* permissions for directories to support append-only files and directories. SELinux also defines a *reparent* permission for directories that controls whether the parent directory link can be changed by a *rename*.

SELinux provides control over each object affected by a file or directory service. For example, in addition to

PERMISSION(S)	DESCRIPTION
read	Read
write	Write or append
append	Append
poll	Poll/select
ioctl	IO control
create	Create
execute	Execute
access	Check accessibility
getattr	Get attributes
setattr	Set attributes
unlink	Remove hard link
link	Create hard link
rename	Rename hard link
lock	Lock or unlock
relabelfrom relabelto transition	Relabel

Table 4: Permissions for the pipe and file object classes.

checking access to the parent directory, SELinux defines permissions for controlling access to the individual file itself for operations such as *stat*, *link*, *rename*, *unlink*, and *rmdir*.

3.3 Socket Controls

SELinux provides control over socket IPC through a set of layered controls over sockets, messages, nodes, and network interfaces. Currently, the SELinux prototype only provides labeling and controls for_INET and UNIX domain sockets. At the socket layer, SELinux controls the ability of processes to perform operations on sockets. At the transport layer, SELinux controls the ability of sockets to communicate with other sockets. At the network layer, SELinux controls the ability to send and receive messages on network interfaces, and it controls the ability to send messages to nodes and to receive messages from nodes. SELinux also controls the ability of processes to configure network interfaces and to manipulate the kernel routing table.

Since sockets are accessed through file descriptions, the socket object classes inherit the permissions defined for controlling access to the file object classes. Only a subset of these permissions are meaningful for sockets. Table 6 shows additional permissions that are specifically defined for controlling access to the socket object classes. The connection-oriented service provided by stream sockets requires several additional permissions, as shown in Table 7. Permissions for network interfaces and nodes are shown in Table 8.

Sockets effectively serve as communication proxies for processes in the SELinux control model. Consequently, sockets are labeled with the label of the creating process by default. A process may create and use a socket with a different label to perform socket IPC with

PERMISSION(S)	DESCRIPTION
add_name	Add a name
remove_name	Remove a name
reparent	Change parent directory
search	Search
rmdir	Remove
mounton mountassociate	Use as mount point

Table 5: Additional permissions for the directory object class.

PERMISSION(S)	DESCRIPTION
bind	Bind name
name_bind	Use port or file
connect	Initiate connection
getopt	Get socket options
setopt	Set socket options
shutdown	Shut down connection
recvfrom	Receive from socket
sendto	Send to socket
recv_msg	Receive message
send_msg	Send message

Table 6: Additional permissions for the socket object classes.

PERMISSION(S)	DESCRIPTION
listen	Listen for connections
accept	Accept a connection
newconn	Create new socket for connection
connectto	Connect to server socket
acceptfrom	Accept connection from client socket

Table 7: Additional permissions for the TCP and Unix stream socket object classes.

PERMISSION(S)	DESCRIPTION
getattr	Get attributes
setattr	Set attributes
tcp_recv	Receive TCP packet
tcp_send	Send TCP packet
udp_recv	Receive UDP packet
udp_send	Send UDP packet
rawip_recv	Receive Raw IP packet
rawip_send	Send Raw IP packet

Table 8: Permissions for the network interface and node object classes.

a different source security label. A process may set up a listening socket so that server sockets created by connections are labeled with either a specified label or with the label of the connecting client socket to act as a server for multiple labels.

SELinux allows the security policy to distinguish between clients and servers for stream socket connections through the *connectto* and *acceptfrom* permissions. SELinux allows the security policy to base decisions on the kind of socket through the use of object classes, and it allows the security policy to base decisions on the message protocol through the per-protocol node and network interface permissions.

SELinux provides control over the association between INET domain sockets and port numbers and the association between UNIX domain sockets and files. Hence, the security policy can restrict the use of port numbers and pathnames for use by particular processes. SELinux also provides control over open file description transfer via UNIX domain sockets.

In SELinux, messages are associated with both the label of their sending socket and a separate message label. By default, this message label is the same as the sending socket label. A process may explicitly label individual messages if the underlying protocol supports message boundaries, i.e. datagram sockets. Messages sent on a stream socket all have the same label, which is the label of the stream socket.

Support for communicating message labels across the network has not yet been implemented in SELinux. The Fluke implementation of the Flask architecture used IPSEC/ISAKMP both to label and protect messages, storing the labeling information in the IPSEC security association. During an ISAKMP negotiation, the appropriate security contexts are sent across the network and the peer obtains SIDs for these security contexts and stores them in its IPSEC security association. When messages are subsequently received that use the IPSEC security association, the messages are validated and then labeled with the SIDs from the association. Similar support will be provided in SELinux using the FreeSWAN [14] IPSEC implementation. Integrating FreeSWAN with the SELinux network mandatory access controls is the next major phase for SELinux development.

4 Application Programming Interface

Typically, the SELinux mandatory access controls operate transparently to applications and users. The labeling decisions of the Flask architecture provide appropriate default behaviors so that the existing Linux application programming interface (API) calls can be left unchanged. The mandatory access controls are only vis-

ible to applications and users upon access failures, in which case they return the normal Linux error codes (e.g. *EACCES*, *EPERM*, *ECONNREFUSED*, *ECONNRESET*) for such failures. In most cases, the potential for these same error conditions already existed with the ordinary Linux kernel, so most applications should handle these conditions. Only a few controls, such as the controls on individual *read* and *write* calls, can cause access failures where an access failure was not previously possible.

Although existing applications can be used unmodified, it is desirable to provide new API calls to allow modified and new applications to be developed that have some degree of awareness of the new security features. Each SELinux kernel subsystem provides a set of new API calls that extend existing API calls with additional parameters for SIDs. The process management subsystem provides calls to get the current and old SIDs of a process, and a call to execute a program with a specified SID. The filesystem subsystem provides calls to create files with particular SIDs, calls to obtain the SIDs of files and filesystems, and calls to change the SIDs of files and file systems. The socket IPC subsystem provides calls to create sockets and messages with particular SIDs, calls to obtain the SIDs of sockets and messages, and calls to specify the desired SID for peer sockets. The same set of controls used for the existing API calls are also applied to these extended API calls, with the only difference being the use of an application-provided SID rather than a default SID.

Applications that use these new calls need to be able to convert between SIDs and security contexts. Furthermore, it is desirable to allow applications to obtain security policy decisions from the security server so that security policies can be defined that control access to application abstractions. For example, a windowing system might be enhanced to provide labeling and separation of windows, with controlled cut-and-paste between windows, or a database system might be enhanced to provide labeling and separation of individual database records maintained in a single file. Such application policy enforcers would still be controlled by the kernel mandatory access controls but could further refine the granularity of protection provided by the kernel. To support such applications, the security server provides a set of new API calls that export its services for converting between SIDs and contexts and obtaining security policy decisions. A set of controls is defined for these new API calls to ensure that the policy can control the ability to use them. An application access vector cache library could easily be created based on the SELinux kernel access vector cache implementation to provide security decision caching for applications.

5 Security Policy Configuration

This section describes the example security policy configuration that has been developed for the Security-Enhanced Linux. At a high level, the goals of the example security policy configuration are to demonstrate the flexibility and security of the mandatory access controls and to provide a simple working system with minimal modifications to applications. The example security policy configuration consists of a combination of Role-Based Access Control (RBAC) [11] and Type Enforcement [8]. The configuration draws from the Domain and Type Enforcement (DTE) configuration described in [26], although it uses a different configuration language described in [16].

The example security policy configuration defines a set of Type Enforcement domains and types. Each process has an associated domain, and each object has an associated type. The policy configuration specifies the allowable accesses by domains to types and the allowable interactions among domains. It specifies what program types can be used to enter each domain and the allowable transitions between domains. It also specifies automatic transitions between domains when certain program types are executed. These transitions ensure that system processes and certain programs are placed into their own separate domains automatically.

The configuration also defines a set of roles. Each process has an associated role. All system processes run in the *system_r* role. Two roles are currently defined for users, *user_r* for ordinary users and *sysadm_r* for system administrators. These user roles are initially set by the *login* program and can be changed by a *newrole* program similar to the *su* program.

The policy configuration specifies the set of domains that can be entered by each role. Each user role has an associated initial login domain, the *user_t* domain for the *user_r* role and the *sysadm_t* domain for the *sysadm_r* role. This initial login domain is associated with the user's initial login shell. As the user executes programs, transitions to other domains may automatically occur to support changes in privilege. Often, these other domains are derived from the user's initial login domain. For example, the *user_t* domain transitions to the *user_netscape_t* domain and the *sysadm_t* domain transitions to the *sysadm_netscape_t* domain when the *netscape* program is executed to restrict the browser to a subset of the user's permissions.

Figure 4 shows a portion of the policy configuration that allows the administrator domain (*sysadm_t*) to run the *insmod* program to insert kernel modules. The *insmod* program is labeled with the *insmod_exec_t* type and runs in the *insmod_t* domain. The first rule allows the *sysadm_t* domain to run the *insmod* program. The sec-

```
allow sysadm_t insmod_exec_t:file x_file_perms;
allow sysadm_t insmod_t:process transition;
allow insmod_t insmod_exec_t:process { entry-
point execute };
allow insmod_t sysadm_t:fd inherit fd_perms;
allow insmod_t self:capability sys_module;
allow insmod_t sysadm_t:process sigchld;
```

Figure 4: Configuration for running *insmod*.

ond rule allows the *sysadm_t* domain to transition to the *insmod_t* domain. The third rule allows the *insmod_t* domain to be entered by the *insmod* program and to execute code from this program. The fourth rule allows the *insmod_t* domain to inherit and use file descriptors from the *sysadm_t* domain. The fifth rule allows the *insmod_t* domain to use the *CAP_SYS_MODULE* capability. The last rule allows the *insmod_t* domain to send the *SIGCHLD* signal to *sysadm_t* when it exits.

From this small portion of the policy configuration, it is clear that the flexibility of the mandatory access controls also yields a corresponding increase in the complexity of managing the security policy. Creating and maintaining a configuration to meet a set of security requirements and verifying that the configuration is consistent with those requirements can be a challenging task. In order for SELinux to be widely deployed and used, a collection of base policy configurations must be developed to meet common sets of security requirements to allow its use by end users with no security expertise. Furthermore, higher-level configuration languages and policy analysis tools are needed to address these challenges.

The security policy configuration controls various forms of raw access to data. The policy configuration defines distinct types for kernel memory devices, disk devices, and */proc/kcore*. It defines separate domains for processes that require access to these types, such as *klogd_t* and *fsadm_t*.

The configuration protects the integrity of the kernel. The policy configuration defines distinct types for the boot files, module object files, module utilities, module configuration files and *sysctl* parameters, and it defines separate domains for processes that require write access to these files. As illustrated by the example in Figure 4, the configuration defines separate domains for the module utilities, and it restricts the use of the module capability to these domains. It only allows a small set of privileged domains to transition to the module utility domains.

The integrity of system software, system configuration information and system logs is protected by the configuration. The policy configuration defines distinct

types for system libraries and binaries to control access to these files. It only allows administrators to modify system software. It defines separate types for system configuration files and system logs and defines separate domains for programs that require write access.

The configuration confines the potential damage that can be caused through the exploitation of a flaw in a process that requires privileges, whether a system process or privilege-enhancing (setuid or setgid) program. The policy configuration places these privileged system processes and programs into separate domains, with each domain limited to only those permissions it requires. Separate types for objects are defined in the policy configuration as needed to support least privilege for these domains.

Privileged processes are protected from executing malicious code. The policy configuration defines an executable type for the program executed by each privileged process and only allows transitions to the privileged domain by executing that type. When possible, it limits privileged process domains to executing the initial program for the domain, the system dynamic linker, and the system shared libraries. The administrator domain is allowed to execute programs created by administrators as well as system software, but not programs created by ordinary users or system processes.

The configuration ensures that the administrator role and domain cannot be entered without user authentication. The policy configuration only allows transitions to the administrator role and domain by the `login` program, which requires the user to authenticate before starting a shell with the administrator role and domain. It prevents transitions to the administrator role and domain by remote logins to prevent unauthenticated remote logins via `.rhosts` files. A `newrole` program was added to permit authorized users to enter the administrator role and domain during a remote login session, and this program re-authenticates the user. To provide confidentiality of secret authentication information, the policy configuration labels the shadow password file with its own type and restricts the ability to read this type to authorized programs such as `login` and `su`.

Ordinary user processes are prevented from interfering with system processes or administrator processes. The policy configuration only allows certain system processes and administrators to access the `procfs` entries of processes in other domains. It controls the use of `ptrace` on other processes, and it controls signal delivery between domains. It defines separate types for the home directories of ordinary users and the home directories of administrators. It ensures that files created in shared directories such as `/tmp` are separately typed based on the creating domain. It defines separate types for terminals

based on the owner's domain.

The configuration protects users and administrators from the exploitation of flaws in the `netscape` browser by malicious mobile code. The policy configuration places the browser into a separate domain and limits its permissions. It defines a type that users can use to restrict read access by the browser to local files, and it defines a type that users can use to grant write access to local files.

6 Performance

This section discusses the impact of the SELinux security mechanisms on the performance of the Linux kernel. The set of benchmarks used was influenced by the *Linux Benchmarking HOWTO* [6]. Microbenchmark tests were performed to determine the performance overhead due to the SELinux changes for various low-level system operations. Macrobenchmark tests were performed to determine the impact of the SELinux changes on the performance of typical workloads.

Each test was performed with two different kernel configurations. The *base* kernel configuration corresponds to an unmodified Linux 2.4.2 kernel. This configuration was measured to provide the performance baseline for each benchmark. The *selinux* configuration corresponds to an enforcing Security-Enhanced Linux 2.4.2 kernel. The performance measurements of the *selinux* configuration can be compared against the baseline to determine the overhead imposed by the SELinux security mechanisms.

6.1 Microbenchmarks

The microbenchmark tests were drawn from the UnixBench 4.1.0 benchmark [21] and the `lmbench` 2 benchmark [18] suites. These microbenchmark tests were used to determine the performance overhead of the SELinux changes for various process, file, and socket low-level operations. These benchmarks were executed on a 333MHz Pentium II with 128M RAM. The `lmbench` network tests ran server programs on a 166MHz Pentium with 64MB RAM. Both the client and server machines ran the same kernel for the `lmbench` network benchmarks so that the results show the total cost of the SELinux overhead on both systems.

6.1.1 UnixBench The results for the UnixBench system microbenchmarks are shown in Table 9. The file copy benchmark measures the rate at which data can be transferred from one file to another, using various buffer sizes. For small buffer sizes, the system call overhead dominates the time to copy the file. The SELinux overhead consists of revalidating permissions for each read and write for the file copy. As the buffer size increases,

Microbenchmark	Base	SELinux	Overhead
file copy 4KB	49.5	48.6	2%
file copy 1KB	40.4	38.6	5%
file copy 256B	23.0	21.0	10%
pipe	6.17	7.17	16%
pipe switching	12.7	15.0	18%
process creation	485	494	2%
execl	2480	2610	5%
shell scripts (8)	659	684	4%

Table 9: UnixBench system microbenchmarks. File copy throughput is in megabytes per second. The other UnixBench microbenchmarks are in microseconds per loop iteration (or milliseconds for the shell scripts benchmark). These results were converted into units that can be more easily compared with the lmbench results.

the time to copy the file becomes dominated by the unaffected memory copying costs, so the SELinux overhead becomes negligible.

The pipe benchmark measures the number of times a process can write 512 bytes to a pipe and read them back per second. The pipe switching benchmark measures the number of times two processes can exchange an increasing integer through a pipe. The SELinux overhead consists of revalidating permissions for each read and write on the pipe.

The process creation test measures the number of times a process can fork and reap a child that immediately exits. The SELinux overhead consists of performing a permission check on each fork and wait operation. The execl benchmark measures the number of execl calls that can be performed per second. The SELinux overhead consists of computing the label for the transformed process and performing permission checks for searching the path, executing the program, and inheriting open file descriptions.

The shell scripts test measures the number of times per minute a process can start and reap a set of 8 concurrent copies of a shell script, where the shell script applies a series of transformations to a data file. The SELinux overhead consists of computing the label for processes for each program execution, computing the label for new files created by the scripts, and performing permission checks for the various process and file operations.

6.1.2 lmbench The results for the lmbench microbenchmarks are shown in Table 10. The null I/O benchmark measures the average of the times for a one-byte read from `/dev/zero` and a one-byte write to `/dev/null`. The SELinux overhead consists of revalidating permissions on each read and write.

The stat benchmarks measures the time to invoke the `stat` system call on a temporary file. The SELinux overhead consists of performing permission checks for searching the path and obtaining the file attributes. The

Microbenchmark	Base	SELinux	Overhead
null I/O	1.45	1.93	33%
stat	8.06	10.3	28%
open/close	11.0	14.0	27%
0KB create	22.0	26.0	18%
0KB delete	1.72	1.90	10%
fork	499	505	1%
execve	2730	2820	3%
sh	10K	11K	10%
pipe	12.5	14.0	12%
AF_UNIX	20.6	24.6	19%
UDP	310	356	15%
RPC/UDP	441	519	18%
TCP	389	425	9%
RPC/TCP	667	726	9%
TCP connect	675	738	9%

Table 10: lmbench microbenchmarks. Measurements are in microseconds. Measurements below the bar represent round-trip latency for various forms of IPC.

open/close test measures the time to open a temporary file for reading and immediately close it. The SELinux overhead consists of performing permission checks for searching the path and opening the file with read access.

The 0K create and 0k delete benchmarks measure the time required to create and delete a zero-length file. For the 0K create, the SELinux overhead consists of computing the label for the new file and performing permission checks for searching the path, modifying the directory, and creating the file. The SELinux overhead for the 0K delete consists of performing permission checks for searching the path, modifying the directory, and unlinking the file.

The fork, execve, and sh benchmarks measure three increasingly expensive forms of process creation: fork and exit, fork and execve, and fork and execlp of the shell with the new program as a command to the shell. For the fork benchmark, the SELinux overhead consists of permission checks on fork and wait, as with the UnixBench process creation benchmark. For the execve benchmark, the SELinux overhead consists of the fork overhead plus the label computation and permission checks associated with program execution, as with the UnixBench execl benchmark. For the sh benchmark, this overhead is further increased by the additional layer of process creation, program execution, and path searching by the shell.

The remaining lmbench tests measure round-trip latency in microseconds for various forms of interprocess communication between a pair of processes. The lmbench bandwidth benchmark results are omitted since they did not show any significant difference between the *base* and *selinux* configurations, as expected.

The SELinux overhead on the pipe benchmark consists of revalidating permissions on each read and write,

as with the UnixBench pipe switching benchmark. For the AF_UNIX benchmark, the SELinux overhead consists of checking permission to each socket and revalidating the permissions for the connection between the sockets on each send and receive. For each of the networking benchmarks, the SELinux overhead includes checking permission to each socket, host, and network interface for each packet. The overhead for the UDP and RPC/UDP benchmarks also includes checking permission between the socket pair on each send and receive. For the TCP and RPC/TCP benchmarks, SELinux revalidates the permissions granted during connection establishment between the socket pair on each send and receive. The SELinux overhead for the TCP connection benchmark includes the permission checks between the socket pair for the connection on connect and accept.

6.1.3 Conclusion Although the percentage overhead for some of the microbenchmark results is large, the real difference in absolute times is typically quite small and becomes insignificant for macro operations, as shown by the results in Section 6.2. Furthermore, these results must be viewed as an upper bound on the performance overhead, since neither the AVC nor the security server implementation have been optimized. Other known areas where the performance could be improved include making better use of AVC entry references and improving the AVC locking scheme.

6.2 Macrobenchmarks

The first macrobenchmark consisted of compiling the Linux 2.4.2 kernel sources, since this involves significant file system activity and is representative of a workload experienced commonly by Linux users. The second macrobenchmark was the WebStone 2.5 benchmark for web servers [19], which is representative of a typical workload for a web server.

For the kernel compilation macrobenchmark, the time to execute “make” was measured. The 2.4.2 kernel sources were configured with the default options, and a “make dep” was done prior to the testing. Three kernel compilations were performed, each immediately after a reboot into single-user mode, and the results were averaged. This benchmark was executed on a 333MHz Pentium II with 128M RAM.

For the WebStone macrobenchmark, one hundred 10-minute trials were run with 32 web clients requesting the standard WebStone file set. A Sun Ultra 5 running SunOS 5.6 with 128M RAM was used as the test controller and client machine. This machine was directly connected using a 10Mbit Ethernet crossover cable to a 133MHz Pentium with 64M RAM running the Apache web server provided with RedHat 6.1.

	Base	SELinux	Overhead
elapsed	11:14	11:15	0%
system	00:49	00:51	4%
latency	0.56	0.56	0%
throughput	8.29	8.28	0%

Table 11: Macrobenchmark results. The elapsed and system times for a “time make” on the Linux 2.4.2 kernel sources are shown in minutes and seconds. The latency in seconds and throughput in Mbits per second are shown for the WebStone benchmark.

Table 11 displays the results of the macrobenchmarks. There was no significant change in the total elapsed time, and there was only a 4% increase in the system time for a kernel compilation. There was no significant change in either the latency or the throughput measurements for WebStone. At the macro level, there appears to be little noticeable difference.

7 Related Work

The project that is most similar to SELinux is the Rule Set Based Access Control (RSBAC) [22] for Linux project. RSBAC is based on the Generalized Framework for Access Control (GFAC) [4]. Like the Flask architecture, the GFAC separates policy from enforcement and can support a variety of security policies. RSBAC provides a Role Compatibility policy module that is very similar to the SELinux Type Enforcement policy module.

However, RSBAC also differs from SELinux in a number of ways. The GFAC does not specifically address the issue of atomic policy changes, so RSBAC lacks the SELinux support for dynamic security policies. Since the GFAC places the responsibility for managing security labels in its Access Control Information (ACI) module, RSBAC does not provide policy-independent data types for security labels. The RSBAC Access Decision Facility (ADF) depends on kernel-specific data structures, and RSBAC does not provide a security decision cache mechanism, because the RSBAC ADF was directly implemented as a kernel subsystem. In contrast, since SELinux’s predecessor systems implemented the security server as a user-space server running on a microkernel, the SELinux security server is cleanly decoupled from the kernel and SELinux provides the access vector cache.

Since RSBAC was not designed with security-aware applications and application policy enforcers in mind, it lacks equivalents for the extended API calls and new API calls of SELinux, only providing calls for setting and getting attributes of existing subjects and objects. RSBAC uses the Linux real user identity attribute for its decisions and must control changes to this attribute, so it is not completely orthogonal to the existing Linux access

controls. Finally, RSBAC lacks a number of the controls provided by SELinux for each of the kernel subsystems.

Type Enforcement [8] (TE) and Domain and Type Enforcement (DTE) [5] have a number of similarities to SELinux, since SELinux provides a generalization of TE in its example security server. Two projects are integrating DTE into Linux [15, 1]. SELinux was designed to provide flexible support for a variety of policy models, while DTE was only designed to implement an enhanced form of TE. DTE is distinguished from traditional TE by the DTE Language (DTL) for expressing access control configurations and by an implicit typing mechanism based on the directory hierarchy for labeling files. The SELinux TE policy module likewise has a configuration language for expressing access control rules. SELinux stores file labels explicitly, but allows labels to be managed using a higher-level specification based on pathname regular expressions. NAI Labs' DTE prototype also provided labeling and controls for NFS and was integrated with IPSEC.

The TrustedBSD project is developing a variety of trusted operating system features, including mandatory access controls, for FreeBSD [27]. SELinux differs from TrustedBSD in that SELinux is a more mature system, that it addresses only mandatory access controls, and that it uses a flexible mandatory access control architecture rather than hardcoded policies. The TrustedBSD project plans to migrate to a more flexible mandatory access control architecture in the future [28].

The Medusa DS9 [3] project is similar to SELinux at a high level in that it is also developing a kernel access control architecture that separates policy from enforcement. However, Medusa is very different in its specifics. In Medusa, the kernel consults a user-space authorization server for access decisions. The Medusa access controls are primarily based on labeling subjects and objects with sets of virtual spaces to which they belong and defining what virtual spaces can be seen, read, and written by each subject. The authorization server can also require explicit authorization in addition to the virtual space checking, in which case it can apply other kinds of policy logic and can even override the ordinary Linux access controls. Medusa DS9 also provides support for system call interception by the authorization server and for forcing a process to execute code provided by the authorization server.

The Linux Intrusion Detection System (LIDS) [2] provides a set of additional security features for Linux. It supports administratively-defined access control lists for files that identify subjects based on their program. Like Medusa, LIDS can control the ability to see files and processes in directory listings. LIDS also supports defining capability sets for programs, preventing certain

processes from being killed, sending security alerts on access failures, and detecting port scans.

The LOMAC [13] project has implemented a form of mandatory access control based on the Low Water-Mark model in a Linux loadable kernel module. LOMAC was not designed to provide flexibility in its support for security policies; instead, it focuses on providing useful integrity protection without any site-specific configuration, regardless of the software and users present on a system. It should be possible to implement the Low Water-Mark model in SELinux as a particular policy module.

8 Summary

This paper explains the need for mandatory access control (MAC) in mainstream operating systems and presents the NSA's implementation of a flexible MAC architecture called Flask in the Security-Enhanced Linux (SELinux) prototype. The paper explains how the Flask architecture separates policy from enforcement and provides the necessary interfaces and infrastructure for flexible policy decisions and policy changes. It describes the fine-grained labeling and controls provided by SELinux for kernel objects and services. The paper explains how existing Linux applications can run unchanged on the SELinux kernel, and it describes the support for security-aware applications. The paper shows how the SELinux controls can be applied to meet real security objectives by describing the example security policy configuration. It demonstrates that the performance overhead of the SELinux controls is minimal. Finally, the paper highlights the differences between SELinux and related systems.

Availability

The Security-Enhanced Linux software is available under the GNU General Public License (GPL) at <http://www.nsa.gov/selinux>.

Acknowledgments

We thank Timothy Fraser for his contributions to the example policy configuration and for his assistance in porting the kernel modifications to the 2.4 kernel. We thank Anthony Colatrella and Timothy Fraser for assisting with the performance benchmarking and analysis. We also thank Ted Faber, Timothy Fraser and the anonymous reviewers for reviewing earlier drafts of this paper.

References

- [1] Configurable Access Control Effort. <http://research-cistw.saic.com/cace>.

- [2] Linux Intrusion Detection System. <http://www.lids.org>.
- [3] Medusa DS9. <http://medusa.fornax.sk>.
- [4] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson. A Generalized Framework for Access Control: An Informal Description. In *Proceedings of the Thirteenth National Computer Security Conference*, pages 135–143, Oct. 1990.
- [5] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, May 1995.
- [6] A. D. Balsa. Linux Benchmarking HOWTO, Aug. 1997. <http://www.linuxdoc.org/HOWTO/Benchmarking-HOWTO.html>.
- [7] D. E. Bell and L. J. La Padula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.
- [8] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.
- [9] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.
- [10] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, Apr. 1987.
- [11] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Oct. 1992.
- [12] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.
- [13] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.
- [14] J. Gilmore. FreeSWAN. <http://www.freeswan.org>.
- [15] S. Hallyn and P. Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Oct. 2000.
- [16] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. Technical report, NSA and NAI Labs, Oct. 2000.
- [17] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Oct. 1998.
- [18] L. McVoy and C. Staelin. Imbench 2. <http://www.bitmover.com/-lmbench>.
- [19] MindCraft. Webstone. <http://www.mindcraft.com/webstone>.
- [20] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.
- [21] D. C. Niemi. Unixbench 4.1.0. <http://www.tux.org/pub/tux/-niemi/unixbench>.
- [22] A. Ott. Rule Set Based Access Control as proposed in the Generalized Framework for Access Control approach in Linux. Master's thesis, University of Hamburg, Nov. 1997. pp. 157. <http://www.rsbac.org/papers.htm>.
- [23] Secure Computing Corp. DTOS Formal Security Policy Model. DTOS CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, Sept. 1996. <http://www.securecomputing.com/randt/HTML/-dtos.html>.
- [24] Secure Computing Corp. DTOS Generalized Security Policy Specification. DTOS CDRL A019, 2675 Long Lake Rd, Roseville, MN 55113, June 1997.
- [25] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, Aug. 1999.
- [26] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining Root Programs with Domain and Type Enforcement. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, California, 1996.
- [27] R. Watson. Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD. In *Proceedings of the 2000 BSD Conference and Expo*, Oct. 2000.
- [28] R. Watson. Robert Watson on FreeBSD and TrustedBSD, Jan. 2001. <http://slashdot.org/interviews/01/01/18/1251257.shtml>.

A Practical Scripting Environment for Mobile Devices

Brian Ward

Department of Computer Science

The University of Chicago

bri@cs.uchicago.edu

Abstract

Software development for small mobile devices concentrates on cross-compiling C into platform-specific object code. Each platform has its own idiosyncrasies in setting up an application, memory management, display management, and so on. We present a new programming approach based on dynamic content generation that simplifies development of common applications for mobile devices. This paper introduces new software development tools intended for mobile devices; our approach utilizes a compiler and a multiplatform runtime environment. In the text, we first note a few problems in mobile code development, then briefly analyze a typical mobile application. From this, we draw a similarity to a familiar programming model, and formulate what we would like to borrow from that and apply to mobile code development. Finally, we present the current status of the project and its future plans.

1 Introduction

When a programmer gets a new toy like a Palm handheld, one of the first things that comes to mind is “I’d like to write some code for this.” “I’d like to port *foo* to this” also comes to mind. However, the harsh reality of mobile devices is that software development really isn’t very convenient. A typical situation is that one writes the code in C on a workstation, runs a cross-compiler on it, loads the resulting binary into the device, then crashes the device without knowing very well what happened. For the more popular devices, there are some additional tools such as device emulators, but of course, this doesn’t stop the code from crashing. These facilities also do little to help figuring out the obscure ritual required to print “Hello, World” nor do they help in porting the code to other mobile platforms.

Graphical display on mobile devices, as with many environments, isn’t convenient to do on a pixel-by-pixel basis. Palm development in C addresses this in part with ‘forms,’ [8] which are specifications for widget placement on the screen which the programmer pre-specifies separately. Unfortunately, the consequence is that it impedes any dynamic flavor of the display. This is fairly serious, since a mobile device isn’t usually used to do any real computation—it is usually there only to extract and display information.

There are some alternatives to the C model, such as Java2ME[3], but these have not been terribly popular because, for the most part, they have many of the same problems (only the language has changed).

Most programmers would probably think highly of a language like Python being available on the Palm. However, there must be more to it than this, since that has already been done[10]. Not only is there is an acute lack of support for display techniques, but since they were written for a much larger environment such as Unix, they often have to be considerably stripped down, with additional limitations placed on what they can actually do.

Moreover, there is a lack of a cross-platform mobile software development ‘for the rest of us.’ Most people write very simple programs and don’t want to have to worry about memory management, heap and record size limitations, and so on.

2 Application Analysis

Perhaps some illumination on this problem lies not in the means to the end, but in the end itself. One should look at the actual applications which run on these devices and see how they work and what they do.

Certain scripting languages like Python and Perl are popular in part because they rather fit in quite well with the shell interface of a Unix system. When at a full-size machine, command-line batch processing prevails. As we develop scripts, we run them over and over again at our shell prompt, often redirecting the output to other files, where we may look at them with a pager program, a web browser, etc. When satisfied with the output, we either continue to run the scripts at a shell prompt or automate them in some way (using cron jobs, window manager macros, dynamic web pages, and so on).

Our observations show that mobile applications have a much different interface model than those of the typical Unix desktop. When a user runs an application on a handheld, some sort of full-screen display usually appears—often, a menu of sorts. Then one selects something that looks interesting and the page either redisplay, or a new page appears to take the place of the current. In a sense, it's a “tap, read screen, tap, read screen” sort of model. Because this sounds a lot like “point and click,” we have an idea of where to head to next.

3 We've Seen This Before

This interface model is remarkably similar to a popular strategy for larger web sites, that of so-called ‘dynamic content.’ While the actual implementation language and detail vary, most of these web sites boil down to code like this:

```
<? print_header(); ?>
Here's some stuff:<br>
<?
    *fd = db_open("server", "dbname");
    $result = db_query($fd,
        "select * from foo where bar = baz");
```

```
while ($row = db_fetchrow($result)) {
    print $row[2] . " " . $row[1];
    print "<br>\n";
}
db_close($fd);

print_footer();
?>
```

This particular segment uses the PHP[9] language between the `<? ?>` delimiters. The `db*` functions are database access functions.

Most popular mobile applications do the same sort of thing; this likeness is reinforced by the number of personal organizer and directory web sites available (some of which can synchronize with mobile devices).

Though they differ in syntax, implementation, and style, the languages used on the web have one thing in common. In addition to a set of built-in functions, they each offer an interface to add additional functions written in C. This model has proved popular enough to inspire additional tools (e.g. SWIG[2]) meant to further bridge the gap. Due to this level of support, the ‘core’ set of functions grows as the language matures. This leaves the question of what the core functions are. For example, Python aims for a smaller core, while a PHP build attempts to compile and include as many functions as possible up front. Perl lies somewhere in between.

4 Intent

We therefore asked a question on what it take to bring this sort of convenience to mobile platforms. Previous research shows that mobile devices are inherently poorly-connected and resource-poor[11], so we cannot rely on proxy servers to do the computation for us in real-time (the WAP model[12]), nor can we simply start up a web server on the device and then run a web browser (at least not yet).

An important aspect is ease of presentation. A “Hello World” program should not be much more the text itself. Augmenting this are the usual string

manipulation functions available in most scripting languages, as well as interfaces to important system functions (such as database access). This is one of the most important advantages to Unix-based languages such as Python and Perl in web servers—they not only have handy access to common library calls to get the data that they but they also have significant power and ease in producing any kind of output required.

5 Design Overview

A system for doing just this is currently under development. It consists of two parts: a parser/compiler and virtual machine-interpreter. The former takes representation descriptions along with a C/PHP-like language (currently called HHL) and produces a bytecode for the virtual machine-interpreter (named VL).

The VL execution model is similar to the dynamic web server example, which runs scripts once per click and sends them to the web browser as a simple document. When the user runs a program, the code is fed into VL. The immediate output of this is a description of the on-screen content, which VL then draws on the screen, in the same way that a web browser might display HTML. If a whole new page is desired (for example, we're in an address book listing we tap on an entry to bring up the full entry), the whole process repeats for the new page. However, in a mobile environment, we can change a few of the rules. Since the machine which generated the content also happens to be the one displaying the content, all of the information about is still in place. This allows us to update portions of the screen without a full redispatch.

6 The HHL Language

A procedural language, HHL resembles PHP and C. Its flow control constructs include `if-then-else` and `while`. Variables begin with a dollar sign (e.g. `$stuff`).

Functions are defined as in the following example:

```
function add($arg, $another_arg) {  
    global $scale;  
    return($scale * $arg * another_arg);  
}
```

If the function returns no useful value (like a C `void` function), `return` is not necessary. In addition to these user-defined functions, the runtime environment provides a number of built-in functions, such as `atoi()` for converting strings to integers, and `cellp()`, used display modification. It is acceptable to replace a builtin function with a user-defined function within a single program.

There are two basic data types, strings and integers. In addition, the list type provides array- and list-like functions.

7 Display

There are two stages to the display of information. The first is a display specification which determines the layout of the screen and what initially appears. This corresponds to HTML in the web model, and it employs SGML/HTML-like tags. As in a system such as PHP, one can provide a static markup, with HHL code, or a mixture of both.

Instead of using a complete markup language such as HTML, we only employ a small number of tags. In analyzing mobile applications, one HTML-like element stands out among others: the table. From listboxes to buttons, almost anything can be reduced to a table with a user-defined action for taps on the element. This is a pure code segment, making the display a 2x2 matrix of numbers, 1 to 4:

```
<table c=2>  
<c>1</c><c>2</c>  
<c>3</c><c>4</c>
```

One can insert HHL code to create dynamic content:

```
<table c=2>
```

```
<?
  $i = 1;
  while ($i < 5) {
    print "<c>$i</c>"
  }
?>
```

An important element of the display system is that HHL code can interact with the display device and use the information not only to adapt the current screen geometry, but also to detect how much room is left on the screen given the amount of information already present. This example employs the `rows_left()` function:

```
<table c=2>
<c>Row Number</c><c>Rows Left</c>
<?
  $n = 0;
  while ($i = rows_left()) {
    print "<c>";
    print 'n';
    print "</c><c>";
    print $i;
    print "</c>";
    $n++;
  }
?>
```

8 Actions and Display Updates

Now that we've described the display system's basic infrastructure, we must sort out the details of updates to a page. This is where our system most deviates from the web model. We've described the fact that we need to be more aware of our state. A web browser isn't very strong in this regard, nor should it be, for security, consistency, and other reasons. Typically, when the user clicks on something in a page to update it, the page redisplay, sending some state parameters back to the web server. There are some ways to do non-critical browser-side update features with Javascript, but the use of this isn't widespread due to mixed results.

Our system resembles the Tk toolkit's system[7] to a certain degree, but with some modifications for a compiled language. After VL loads and runs an application, it waits for an event, such as a display tap or network activity. The event triggers an event-

handling function if applicable.

When defining a cell with the `<c>` tag, you may provide a name and action, as in this example:

```
<table c=2>
<c name=example action=tap_me>Tap me.</c>
<c name=display></c>
<?
  function tap_me($name) {
    cellp("display", "0w.")
  }
?>
```

When the user taps on the "Tap me" cell, VL calls `tap_me("example")`. When the function completes execution, the VL returns to the idle-event loop.

9 Implementation

9.1 HHL Compiler

The source code for the system has essentially three pieces: the HHL compiler, which is much like the source for any Unix utility, the platform-independent core of VL, and the platform-specific support file sets for each target. Excepting the lex and yacc constructs, all code is ANSI Standard C.

A straightforward tool with few frills, the compiler generates parse trees and messages them, then generates code in two distinct steps. Optimization leans toward eliminating redundancy and space economy. For example, because operations on strings tend to involve multiple function calls and exercise the memory management system, some of those operations are analyzed. Two sequential `print` statements

```
print $str1 . "</c>";
print "<c>" . $str2;
```

would normally generate around ten instructions total in the target code. By condensing this segment into a single `print` statement, we can save three

instructions. Because this sort of optimization requires certain semantic knowledge of builtin functions, possibly leading to an unnecessarily complex compiler, we aim only for a select few such tweaks.

At the moment, the compiler runs only on Unix platforms; while in distinct components, its primary use is on a desktop machine.

9.2 VL Core

VL's development has been multiplatform from the start, immediately forcing a distinction between the front-end and computational components. Furthermore, to avoid code riddled with `#ifdef` directives, some modules require separate files for certain functions for each platform; the display and events interfaces are two examples.

At first glance, the VL object code resembles that of a normal microprocessor. It has register-like storage locations, comparisons, branching, and other common items. However, the design has a number of amenities meant to simplify compilation from a higher-level language. For example, in addition to a regular execution stack, it has another stack for use by operations and functions, not unlike the stack in very old hardware architectures [1]. This not only makes compiler implementation easier, but greatly reduces object file size.

Also in the interest of object size reduction, the instructions are of variable length. A further assumption is that much of the object code will consist of function execution. Therefore, there is some emphasis on function call setup size, and the complexity of returning a value. The storage location addressing scheme provides fast mapping from a storage location address to the execution stack. Though it is possible to implement nested functions using traditional methods dating from Algol[6], they are currently absent in HHL.

VL enforces types at run time. Each storage location has a type, regardless of any memory manager's involvement. However, the type enforcement is not overly strict. Function arguments do not undergo

typechecking during function execution. For user-defined functions, this is not an issue. Any built-in functions check their arguments as they see fit. For example, since a string operation on a non-string might cause a crash, string manipulation functions should make sure that their arguments are indeed strings. There is also no particular obligation to return run-time errors in the case of a type clash; if `atoi()` gets an integer as its argument instead of a string, it is perfectly acceptable to return that same integer as the result instead of flagging an error.

In addition to the (lack of) obligation for types, a builtin function does not need to check its number of arguments. This has an advantage for functions such as string append. We noted above that we could save some instructions by condensing the two print calls into one. In the original form, there are actually four function calls—two `print()` and two string appends. In the condensed form, there are only one of each, where the string append has four arguments instead of two.

The VL core module processes object code. The front end hands the module a memory pointer to the start of the code. The initialization function analyzes the object header, and determines where execution should start. The header includes the names of all functions called in the object code, offset addresses for program-defined functions, and a string constant table. Following the header is the main program code, and finally, the code of all program-defined functions. The initializer makes a note of where the code begins and ends, in the interest of catching illegal program counter access.

The core module includes the instruction executer. After it reads and runs each instruction, it updates the program counter, and leaves an opportunity for an event, such as a tap on the screen or program interrupt. Allowing these interrupts only between instructions simplifies the system not only because we do not have to worry about half-executed instructions, but also cuts down on the complexity of the interrupt handler for each platform.

9.3 VL Functions

Like the core, the VL builtin function framework is also platform-independent. The builtin function vector is a table of HHL function names paired with real function addresses. For example, if the implementation of a builtin function were a C function `vl_builtin_print` and the HHL name were `print`, the entry in the table would be

```
{ "print", vl_builtin_print }
```

When the VL core initializes a program, it identifies all functions that the compiler assumed to be builtin, and attempts to locate them within the function vector. The core stores this location so that it may call a builtin quickly during program execution. We call this dispatching a function; the builtin function dispatcher routine is also in the

To add a builtin function, one needs to write the function with the prototype `Int32 (Int32 argc, Int32 offset)`, and place the function along with its HHL name into the builtin function vector. `argc` is the number of arguments given when the program calls the function, and `offset` is the stack offset. Here is an example which simply adds two arguments together:

```
Int32 vl_builtin_add2
(Int32 argc, Int32 offset) {
    vl_var i, j;

    /* if num of args not 2, return null */
    if (argc != 2) {
        return(1);
    }
    i = grab_var_with_type(offset);
    j = grab_var_with_type(offset+1);
    s_push(*i.value + *j.value, VL_TYPE_INT)
    return(0);
}
```

We see a number of features from this example. First, we note a safety feature: as long as the dispatcher calls the function with a correct value for `argc`, this function cannot walk off the edge of the execution stack. The assignments to `i` and `j` illustrate how to access a the first and second arguments. Finally, we see that there are two ways to exit from

a builtin function; with a return value of 1, indicating that the function doesn't have a meaningful return value (such as `void` in C), or 0, meaning that the function has put its return value on the operation stack mentioned previously. If there are code branches leaving the stack in different states as above, this is not a problem. The dispatcher puts the stack in order if necessary.

Simple builtin functions such as the one in this example do not need to know about their target architecture. However, there are cases in which function implementations are necessarily different, such as the display interface. One may take several approaches in these cases. If the differences are minor, `#ifdef` preprocessor directives may suffice. If there are large discrepancies, it may be necessary to write each implementation separately, possibly placing them in different source files.

10 Stability

10.1 General

One of our primary concerns is to keep programs from causing a crash. In our case, this is an especially serious matter, mainly due to the operating system support on some of our platforms. Since they tend to be on smaller devices, several years behind current processor computational power specifications, operating system designers must make some sacrifices. Memory management is sometimes weaker, in part due to a lack of hardware support.

But if scripting languages on a Unix desktop don't normally experience segmentation faults (other than running dynamically-loadable modules), we should expect to be capable of the same on a handheld. This need not conflict with our interest in performance. For instance, it is not necessary to do a rigorous check on every bit of code upon initialization, seeing if the number of arguments are correct for each instruction is valid, or if functions are called with matching numbers of parameters. A problem such as an invalid instruction pops up soon enough. Other matters cannot be glossed over. For example, each compiled-in string must be checked to see if it

appears as it is advertised: length matching string header value, NULL character at end.

10.2 Type System

There are a few particular areas of VL to concentrate on; some rely on the stability of others. First, in the VL core, we have a few basic data structures to worry about. We have the execution and operation stacks. A common set of access functions can assure us that we will not run off the borders of these. If we check the stack at each access, and make sure that the stacks see no other accesses outside of this function, we will not have an illegal memory access.

Our type system presents some obstacles. A language such as ML is strongly typed and doesn't need type information once the code is compiled[5]. Not only is our type system a mainly runtime affair, but we have no reason to trust the compiled code, for someone may have inserted a rogue instruction. Therefore, we build our type security up from a few base rules.

The first rule is that VL instructions may only set the types of a limited set of data, and that only when an instruction loads something into a storage area (these are similar to assembly 'Load Immediate' instructions). Let's assume that these two types are numbers and compiled-in strings. Numbers present no difficulty as far as memory is concerned, and any safe operation or function will know to check to see if a number is cause for concern. Compiled-in strings have one precaution. Each such string is ultimately represented by an index to a table created at program initialization. If that index isn't in the table when a VL instruction sets this type, a runtime error occurs.

Furthermore, there is no other way to set a type from an instruction. All instructions which shift data around (e.g. to and from the stacks) work with both data and type at once, keeping them together.

Therefore, with instructions alone, there is no way to have a type mismatch that may cause an op-

eration or function to get bad data. This leaves two questions: First, while this type system is safe, wouldn't one like to have more types? In addition, can the builtin functions handle this?

Both questions may be answered at once. Builtin functions are the key to more types. We've already talked about compiled-in strings; naturally, we would like to have dynamically-allocated strings as well. We can make a new rule: only builtin functions may return the dynamically-allocated string type. If these functions make certain that the string data that they return is 'real,' there is no danger in passing it back to the VL core. No core instructions may change the data without changing the type, and as we already mentioned, they can't change the type to anything but the simple ones.

Assuming that our builtin functions don't have any bugs (and of course, we know that never happens), this system suffices to keep bad data out of the builtins' hands. But is it enough to assure that our language is useful? With strings, it certainly is: the only way one would ever get any string that's not known at compile time is to use a builtin such as a database access function or string append. If we can do similar things with network sockets, we can achieve the same level of sophistication. Of course, we don't let VL instructions play with pointers. While this can hinder us in some programming environments [4], our system is not intended to be a platform for heavy-duty applications.

With this in mind, it is worthwhile to emphasize that builtin functions must dutifully check their arguments and return a valid data-type pair. If one bad piece of data got into the mix, the result could be catastrophic.

One more note about builtin functions and stability is in order, pertaining to blocking operations. On some platforms, blocked operations can interfere with the operating system. If a program is busy with something and doesn't handle an event from the operating system, application buttons may not work, events may be dropped, the system may get sluggish, or worse. Therefore, builtin functions have something of an obligation to try to avoid blocking operations, or if they must be busy for a period of time, to try to service events if they come in. VL

provides an interface for this in its front end through two function calls. One checks for pending events, and the other services any such events.

11 Security Concerns

In a portable development environment, one should explain any security measures because one can easily transfer code (HHL or VL object code) over a network. Our general philosophy towards the matter is the same as for a normal scripting language such as Perl or Python: if you want to run a program you got off the net, you should be a little wary of it. Moreover, this system is not intended as an alternative to the world wide web; it is meant primarily for small application development.

However, there is an extra danger because some potential target platforms do not have the same notions of users and system security as Unix. As a regular Unix user, the most destructive thing you can normally do is to erase all of your own files. The system doesn't care. But on a machine such as a Palm handheld, you can not only erase all of your files, but also the whole system.

One approach to this is just to offer the advice "back up your data." While hard to argue with, not everyone does it, so can we provide more protection? At this point, we offer a proposition on how one might address this issue (no such system has been implemented). Other languages such as Tcl have network safety features when compiled in a certain way; they simply disable builtin functions which may alter the system. For example, we should make databases read-only for an untrusted program, and even turn the network functions off so that the data cannot 'escape' the system.

However, maybe we want to allow selective access, as we'd like our programs to access personal data and do something with it. We can flag any database access and have the user manually confirm it the first time through. Access control lists to certain builtin functions and databases are another possibility. However, this leaves questions of how the user will react to this; they may just develop a habit of

confirming everything that comes their way.

12 Sample Applications

So far we have talked much about the development infrastructure, but little about the actual programs that we intend to write in this system. Therefore, we give some possibilities that are particularly suited to mobile devices.

Custom Schedules. Part of the initial motivation for this project came from the desire for something better than paper train schedules. Various attempts have been made to put train schedules on handhelds, but apparently due to the formatting complexity, these seem just like the paper schedules, only harder to read. An invaluable schedule application would quickly hone in on a certain route, remember it, and perhaps most importantly, automatically find the next trains based on the current time of day.

There are schedules on the web which show some potential features. For example, some transport agencies' sites allow the user to click on a station to bring up information about that station—address, phone, and so on. This would be a simple function for our application to mirror, as it is only a page redisplay.

Network Monitoring. Because there is nothing to stop builtin functions from accessing network data, it should be possible to run network monitoring programs on VL. In addition to the usual socket interface, the creative network administrator may wish to customize their VL, adding specialized builtin functions to filter through a large amount of network data and return only items of interest (remember that a handheld's display isn't large enough to display much).

Simple Games. An unofficial (and not often stated) measure of a development tool is the quality of the games that it can produce. Clearly, VL isn't terribly suited for 3-D animated action games, but it can easily accommodate static games. Examples of these include the minesweeper type games, card games, and the ever-popular DopeWars.

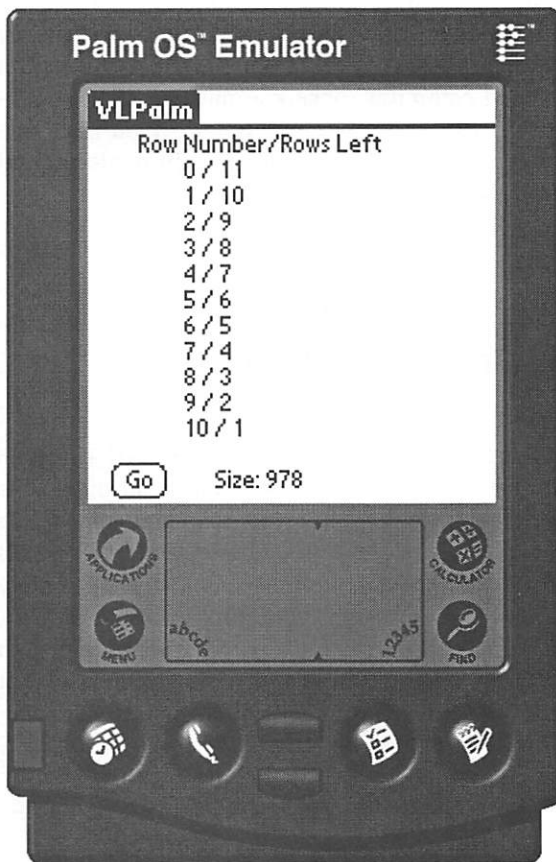


Figure 1: VL on Palm

13 Operation

The HHL compiler runs on Unix-like systems; it acts like any Unix compiler. VL, on the other hand, runs on the PalmOS platform as well as Unix. The compiler is named `hhlc`. Compiling a source file is simple:

```
$ hhlc file.hhl
```

If there were no errors in the source, `hhlc` creates a VL object file called `out`. Simple debugging may be accomplished with a command-line version of VL, simply called `vl`. This version has no graphical capabilities, therefore, it doesn't require a display or any extra hardware. Its primary function is debugging; it reads text from the standard input, and if

the text matches the name of a cell, any defined action for that cell executes. It's therefore possible to run through actions on programs by redirecting standard output.

Unix VL only needs to read the output file that the HHL compiler produces, but because PalmOS is unfriendly to 'alien files,' a Unix conduit is available for the code transfer into a PalmOS database. Once you're happy with the way the program runs on Unix, one may try it out on the Palm. The `install-vlo` program can transfer out to a handheld.

Assuming that the `vl.palm.prc` Palm application which comes with the `hhlc` and VL distribution is present, we're ready to run the program there. Tapping on VLPalm, and then the **Go** button starts up the program. Figure 1 shows a simple program similar to one illustrated in the **Display** section, one that displays the number of rows remaining.

14 Current Status

The HHL compiler currently supports all core language features (flow control constructs, user-defined functions, basic types, and so on). Needing more refinement are areas such as diagnostic output and optimization. At the moment, HHL syntax errors are flagged with the line number of the error, but no real effort is made to attempt to zero in on the actual problem. Similarly, not much care is taken to process the parse trees created by `hhl.y`. If these trees were to be massaged, cleaning up the internal representation of statement blocks, it would be make some optimizations simpler.

VL's frontiers lie primarily on one front: builtin functions. Because the general framework for builtins is in place, making it simple to add a new function, adding these functions is an incremental process. In particular, more string and display routines are in order, as well as networking functions. While it would be interesting to access certain platform specific features, such as interfacing the IR port on a Palm handheld, this is not a high priority. General functionality (and keeping builtins free of bugs) is

the current focus.

In addition to the builtin functions, work is currently underway on the VL ports to some other platforms. Furthermore, one new modification adds a graphic interface to the Unix stdio-based v1.

Finally, more work is needed in application management.

15 Future Plans

After the system stabilizes to a certain point, there are two avenues of development, both of which are of particular interest to the open source community. The first is to port VL to as many devices as possible. At the moment, Palm Computing has a lion's share of the market for small mobile devices, but in addition to the prospect of supporting their competitors by having true cross-platform code, there is also an issue of hardware and software upgrades. Bytecode written by the HHL compiler would (presumably) not be affected by these.

Another area of interest is in getting the HHL compiler itself ported to some VL target platforms. The first advantage to this is that it allows for "true mobile development." One would no longer need to run a compiler on another machine to create new programs; if one so desired, they could write the program on the target device. Moreover it also create some interesting opportunity for interaction and debugging. While playing with a program, it would be possible to compile a new function and insert it into the currently-running program on the fly, as well as get more information about the current state of the program.

Our environment currently focuses its attention to hardware on handheld computers, in particular, those that have a stylus. These devices are best suited for our interaction model (based on screen taps, not pressing buttons). Because the input is uniform, we can concentrate on providing a consistent and portable user interface. However, there are other mobile devices capable of a VL port, such as cellular telephones. One area of future work could be

adapting the input method to support devices without a stylus or touch pad. Furthermore, the display on mobile phones is much smaller than those on a handheld computer, posing a question of how difficult it is to write programs that work on both display types without too many conditionals around the display size.

16 Concluding Words

Current development tools for mobile platforms tend to the somewhat painful side. Quite a bit of effort is required in writing a program in a language such as C or Java, and the result is often that the emphasis shifts away from the information which is ultimately destined for the screen. The HHL compiler and VL runtime environment offer a content-based approach to code development, with a model more like that of the web, with the additional that platform peculiarities can largely be ignored.

17 Availability

The tools are intended to be distributed as open source. The prototype system, which consists of the simple compiler and virtual machine, is available from

<http://www.o--o.net/comp/>

References

- [1] Barton, R. S. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.* (1961), 393-396.
- [2] Beazley, D. M. "SWIG: An easy to use tool for integrating scripting languages with C and C++," *Proc. 4th USENIX Tcl/Tk Workshop*, 129-139.
- [3] Java 2 Micro Edition.
<http://java.sun.com/j2me/>

- [4] Kernighan, B. "Why Pascal is not my favorite programming language," original 1981; appeared in *Comparing and Assessing Programming Languages*, Prentice-Hall, 1984.
- [5] Milner, R. "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, Volume 17, 1978, 349-375.
- [6] Naur, P., Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. "Revised report on the algorithmic language ALGOL 60," *Communications of the ACM*, 6(1) 1963, 1-17.
- [7] Ousterhout, J. K. *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
- [8] Rhodes, N., McKeehan, J. *Palm Programming*, O'Reilly and Associates, 1999.
- [9] PHP Dynamic Hypertext Processor.
<http://www.php.net/>
- [10] Python to Palm Ports.
<http://www.endeavors.com/pipppy/>
<http://www.isr.uci.edu/projects/sensos/python/>
- [11] Satyanarayanan, M. "Mobile information access," *IEEE Personal Communications*, 3(1), February 1996.
- [12] Wireless Application Protocol.
<http://www.wap.com/>

Nickle: Language Principles and Pragmatics

Bart Massey

Computer Science Department

Portland State University

Portland, Oregon USA 97207 0751

bart@cs.pdx.edu, <http://www.cs.pdx.edu/~bart>

Keith Packard

SuSE Inc.

keithp@suse.com, <http://keithp.com>

Abstract

Nickle is a vaguely C-like programming language for numerical applications, useful both as a desk calculator and as a prototyping and implementation language for numerical and semi-numerical algorithms. Nickle abstracts a number of useful features from a wide variety of other programming languages, particularly functional languages. Nickle's design principles and implementation pragmatics mesh nicely to form a language filling a useful niche in the UNIX software environment. The history of Nickle is also an instructive example of the migration of an application from an idea to a freely-available piece of software.

1 Introduction

The past 20 years have seen an explosion of "utility-belt programming languages". Often implemented as true or byte-code interpreters and designed to operate smoothly in the UNIX environment (in the spirit of `sed` [McM78], `AWK` [AWK88], `bc` [MC78a], `dc` [MC78b], and the like), these languages are intended both to address specific classes of tasks and to be usable for general-purpose programming. Other examples include Perl [WCS96], Python [Lut01], Java [GJSB00], various Scheme [CE92] implementations, and ML [MTH90].

During this time frame, the authors have been intermittently involved in the development of a utility-belt programming language initially tailored to scratch-pad-style numerical calculation, and reflecting design principles including:

- Simplicity of design and implementation.
- Separability of concerns, such that language features can be implemented and used independently.
- Use of best-practice programming language technologies.
- Practical problem-solving utility.

The result is a language with a low learning curve for experienced UNIX programmers that allows the integration of offline programs with online calculations in a flexible yet safe notation.

2 Programming With Nickle

A small example (Figure 1) may help give a feel for the Nickle language. Note that the variables `i` and `t` are declared at first use. The language feels much like C, but with some of the bothersome details of declaration and typing optional.

This much code would typically be placed in a file, and read into a running Nickle session ("`>`" is the default Nickle prompt here and throughout)

```
> load "countsum.5c"
```

During reading (which occurs with no perceptible delay) the input file is incrementally compiled into the running session. We can then interactively create a sample array to work with.

```
> v = [5]{1, 2, 3, 4, 5}
1 2 3 4 5
```

```

function countsum(c, v, n) {
    if (c < 0)
        return 0;
    int t = 0;
    for (int i = 0; i < n; i++)
        t += v[i];
    if (t == c)
        return 1;
    if (t < c)
        return 0;
    return countsum(c, v, n - 1) +
        countsum(c - v[n - 1], v, n - 1);
}

```

Figure 1: Function returning the number of ways that the first n elements of the array v can be added to produce c .

The square brackets are the array constructor, and in this case create a dynamically-typed array of 5 elements. The curly braces, as in C, are being used to surround an initializer list which values the array elements $a[0] = 1 \dots a[4] = 5$. Now we can invoke `countsum()` (which is extracted from a Cribbage scoring program).

```

> countsum(15, v, dim(v))
1
> countsum(12, v, dim(v))
2

```

If a new definition is given interactively for `countsum()`, it will scope out the current definition. As expected, the storage for v is allocated automatically, and released when v becomes unreachable.

3 Language Features

Nickle strives to be a simple yet expressive programming language: always a difficult goal. A reasonable number of simple language features may be combined in powerful ways to solve problems.

3.1 Names, Lifetimes, Types and Values

Nickle has a C-like syntax and procedural imperative semantics. In addition to borrowing from other imperative languages like Java and Modula-3 [Nel91], Nickle incorporates a number of useful notions from the functional programming community.

3.1.1 Names

Nickle supports first-class function values with full static scope. The visibility of a name is controlled by its scope, and by its relationship to its namespace. Nickle namespaces are similar to Java modules, except that they are purely syntactic rather than having anything to do (in principle or in current practice) with the filesystem. The closest notion is probably namespaces in C++. In brief, a namespace is opened by a `namespace` declaration. Each name declared in a namespace may have one of three possible visibilities:

public The name should be visible outside the current namespace, and should be automatically imported.

protected The name should be accessible from outside the namespace via an explicit path, but should not be made directly available by import declarations.

(no keyword) The name should be completely inaccessible outside the namespace.

Names in a namespace may be referenced in one of two ways. First, `public` and `protected` names may be accessed via an explicit namespace qualifier `::`. Thus, the name `Foo::bar` refers to the name `bar` in the namespace `Foo`. Second, the `public` names in a namespace may be brought into unqualified scope using an `import` declaration. For some example uses of namespaces see Figure 3 below and the Appendix.

3.1.2 Lifetimes

There are three options for the lifetime of a Nickle name, as opposed to C's `auto` and `static`. This extra control is a natural consequence of the distinction between global functions and nested functions, which is lacking in C. It manifests noticeably in the times at which initialization occurs. The three lifetimes for a Nickle declaration are:

global The lifetime of the named value is the lifetime of the interpreter evaluating the program. Objects declared `global` are initialized once, when the definition of the function containing them is first encountered.

static The lifetime of the named value is the lifetime of the function in which the definition


```

int() function f () {
    int function g () {
        global x = 0;
        return ++x;
    }
    return g;
}

> g = f();
> g()
1
> g()
1
> g = f();
> g()
1

```

Figure 2: A variable `x` with `auto` scope.

occurs. Objects declared `static` are initialized whenever the function containing their definition is evaluated. In the absence of nested scopes, this lifetime is the same as `global`.

auto The lifetime of the named value is the lifetime of the current function. Objects declared `auto` are initialized whenever their definition is evaluated.

The default lifetimes are as in C: `global` for top-level objects, and `auto` for those local to a function. Note that lifetime is different than scope: different function definitions, for example, may have different global objects named `x`.

Figure 2 is useful illustrating the difference between `auto`, `global` and `static` scope. Imagine that the `auto` declaration of Figure 2 was instead a `static` declaration. In this case, `x` would be initialized whenever the definition of `g` was evaluated, and thus the second invocation of `g()` at top-level would return 2. If `x` was instead declared `global`, it would be initialized only once, when the definition of `f` was compiled, and thus the successive invocations of `g()` would return 1, 2, and 3.

3.1.3 Types

Nickle has two sets of type consistency rules. First, any object in the program may be statically typed. The static type system strongly resembles that of C or Java. Syntactically, Nickle types are required to be like Java's optional "left-hand" syntax: type

declarations appear to the left of program objects, and are modified type-by-example. Semantically, there are subtle but significant differences between the type systems of these languages. Nickle allows objects to be explicitly typed `poly`, indicating that the type need not be statically checked. Unlike C, arrays with different sizes are not of different type. Nickle has more types than C, including a disjoint union type and multi-dimensional array types. Structure types obey a subtype relationship over their members. The `void` type is handled slightly differently than in C: in Nickle, there is a single value of type `void`, written as `<>`, allowing the `void` type to interact more smoothly with the rest of the language without significant loss in security. All of this adds up to a type system which provides full security while retaining reasonable expressiveness.

Secondly, all operations are currently checked for type consistency (as well as performing array bounds checking and the like) at runtime. While in principle many of these runtime checks could be removed by static type checking, and others could be hoisted in order to improve performance, runtime type checking is not currently believed to be a performance bottleneck, and the implementation is greatly simplified by this choice. The combination of strong static checking and complete runtime checking does normally mean that program defects will be caught "as early as possible", providing confidence in execution correctness and aiding debugging.

3.1.4 Values

Nickle supports first-class structured values. Any value of any type may be created and used in any legal expression context. This is important for a language designed for a desk calculator. It also allows the programmer to determine which values in the program need to be named, instead of being forced to produce names by the language semantics.

For example, an anonymous array can be created whenever needed, and treated just like any other array object. In this example, a two-element string array is created and then dereferenced:

```

> ((string[*]){ "a", "b" }) [1]
"b"

```

3.2 Implementation Features

The current implementation of Nickle is an interactive byte compiler, in the style of many Scheme

implementations. All expressions and statements typed at the command line are rapidly compiled into an intermediate byte-code representation that is then evaluated in the compiled top-level context. The language is designed to support implementations using offline compilation to native code, for greater efficiency. While Nickle's predecessors were at one time pure interpreters, the current structure of the language would make such an implementation difficult, and offers no obvious advantages.

The statement syntax of Nickle, as with C, is essentially that of infix expressions, lending itself to scratch-pad-style calculation. Full interactive compilation and first-class support for all values means that it is quite easy to interact with and modify Nickle code written online, to develop new code online, and to calculate interactively in this environment. The easy loading and import of appropriate namespaces allows a custom calculation environment to be quickly set up.

Many of the features of Nickle are predicated upon the existence of automatic storage management. The Nickle implementation includes a tracing mark-sweep garbage collector, which is used in two principal ways. First, storage is implicitly allocated for Nickle objects during definition. Second the C code that implements Nickle makes use of the garbage collector as a general-purpose storage allocator. This is facilitated by the non-copying nature of the collector, and by CPP macros which allow easy rooting of the garbage collector in a C frame.

3.3 Numeric Features

To some degree, the reason for the existence of Nickle is the support for infinite-precision integer and rational arithmetic. By default, non-integer quantities are represented as rationals, which ensures that precision will never be lost by the computation. For convenient comparison and for familiarity's sake, rationals behave syntactically like floating-point numbers: they are input and output using a floating decimal point.

To avoid loss of precision due to decimal conversion, both output and input representations of rational numbers support "repeating decimals". For example, the constant `0.1{6}` is the Nickle decimal representation of $1/6$. Using sophisticated techniques based on a factored representation, Nickle is capable of calculating the minimum-length repeat for the decimal representation of an arbitrary rational number. This can be very expensive in some cases, however, so by default repeating representations beyond

a certain number of digits will be truncated. While this may lead to loss of precision on input or output, it can be disabled, and in any case all calculations will still be performed internally to full precision.

Some real numbers (irrational numbers such as $\sqrt{2}$ and transcendental numbers such as $\text{acos}(0)$) are not precisely representable as rational numbers. In order to perform calculations involving these quantities, Nickle provides its own implementation of floating-point arithmetic with user-settable mantissa precision and infinite precision exponents. A sensible numeric type hierarchy and well-defined rules for combining various precisions means that Nickle is generally very good at retaining precision in all numeric calculations.

3.4 Flow Of Control Features

In order to cope with situations where a function should not return normally, either as a result of a semantic error such as division by zero or at the user's request, Nickle provides support for the declaration, generation, and handling of *exceptions*. Exceptions are not first-class objects. They are declared in the style of void functions (including arbitrary typed formal parameters), and are scoped identically. At any point during program execution, any in-scope exception may be thrown via a `raise` statement. A Java-style `try-catch` block allows the handling of raised exceptions. Currently, there is no mechanism for "restarting" a computation which has raised an exception: once an exception has been raised, execution will resume at the nearest applicable dynamically enclosing `catch`, or at the top level if none is found.

In addition to exceptions, Nickle provides first-class "continuations", which capture an execution locus and environment. These are not true continuations, as they do not restore the values of variables modified by assignment between the capture of a continuation and its use: however, the actual semantics supported is both much cheaper to execute and arguably more usable for an imperative language. The closest C analog is `setjmp()/longjmp()` (and indeed, the Nickle equivalents share these names). However, unlike C, a `longjmp()` can occur anywhere, not just in a dynamically enclosing function. In addition, since the Nickle primitives are builtins rather than library functions, variables modified between a `setjmp()` and a `longjmp()` have well-defined values after the `longjmp()`.¹

¹In order to support optimization of separately compiled C programs while providing `setjmp()` and `longjmp()` as li-

```

import Semaphore;

semaphore prod, cons;
int i;

void function ints() {
    while(i < 1000) {
        wait(prod);
        i++;
        signal(cons);
    }
}

void function odd_ints() {
    i = 0;
    prod = new(1);
    cons = new(0);
    thread t = fork ints();
    while(i < 1000) {
        wait(cons);
        if (i & 1)
            printf("%d\n", i);
        signal(prod);
    }
    Thread::join(t);
}

```

Figure 3: An odd-number printer using threads.

3.4.1 Threads

Nickle contains a continuation-based thread system, implemented from scratch entirely in UNIX-portable C code at the byte-interpreter level. Thread scheduling utilizes real-time thread priorities with round-robin scheduling among equal-priority threads. Support is provided for thread synchronization via built-in semaphores or mutex variables.

Threads in Nickle are important for at least three reasons. First, some calculations are most naturally performed concurrently. Consider a simple generate-and-test example like the odd-number example of Figure 3. While in this simple example saving the generator state would be easy, in practice more complicated cases (such as for example a chess move generator) are much easier to write if the generation and testing are performed in separate threads.

brary functions, the C standards allow variables after a `longjmp()` to have any value they might have attained since the corresponding `setjmp()`, except for `volatile` variables.

Secondly, prototyping of parallel algorithms is most easily performed using a language with concurrent features. The concurrency primitives provided by Nickle are very similar to those of low-level parallel systems, allowing prototyping of code like that of Figure 3 above for later translation to a truly parallel environment.

Finally, the construction and use of Nickle itself is made easier by the availability of threads. In particular, the combination of continuations and threads eases the implementation of debugging in the presence of exceptions. Normal command-line Nickle execution is handled by a thread separate from the command parser. When an unhandled exception occurs, the debugger can simply expose the thread state, including the exception continuation, to the user for inspection.

3.4.2 Flow Control Primitives

Some minor modifications to the Nickle language to help support threads, continuations, and exceptions have proven to be particularly convenient. Threads are created by a low-precedence `fork` operator: the semantic is that the expression to which the `fork` operator is applied will be evaluated in the new thread returned by the operator. Any thread can retrieve the result of this computation via the `Thread::join()` built-in function. The principal alternative to making the `fork` operator a syntactic part of the language was to make it a built-in function accepting a closure or continuation. This is much less convenient for the user.

Another important syntactic feature concerns the semantics of operations under locks or other temporary invariants. Java provides a `finally` clause of its `try` block, whose primary purpose is to ensure that the lock is restored. This choice has some unfortunate consequences. Consider typical Java code to execute a function under a lock of some sort, shown in Figure 4. First, this code is syntactically complicated. There are many blocks, and the flow of control is not obvious. Second, even though no `catch` clauses are present the `try` is necessary merely to obtain the `finally` clause. Third, the establishment of the lock is syntactically indistinguishable from the rest of the surrounding code, which means that the language can have no idea whether locks and unlocks are matched, and can provide no assistance with locking errors.

In order to remedy these deficiencies, Nickle separates the invariant-management functionality of

```

if (get_lock(&l)) {
    try {
        locked_operation();
    } finally {
        release_lock(&l);
    }
} else {
    throw new LockException("failed");
}

```

Figure 4: Locking in Java using finally.

```

twixt(get_lock(l); release_lock(l))
    locked_operation();
else
    raise lock_exception("failed");

```

Figure 5: Locking in Nickle using twixt.

try from the exception-handling functionality. The Nickle `twixt` statement is syntactically similar to a C `for` loop, but with two arguments instead of three, and an optional `else` clause. A natural reimplementation of Figure 4 using `twixt` is shown in Figure 5. With this syntax, the `get_lock(l)` and `release_lock(l)` operations are syntactically distinguished, and their correspondence is visually obvious. Equally obvious is the fact that the exception is raised as the result of the failure of `get_lock()`. Finally, the gratuitous `try` clause has been eliminated, and the nesting has been regularized.

As a final feature, consider the situation where a continuation is captured using Nickle's `setjmp()` inside `locked_operation()`. Because the `get_lock(l)` operation is known to be protecting the context in which the continuation is captured, Nickle will re-execute it (!) upon `longjmp()` back to the continuation. This would be difficult or impossible in most languages.

3.5 Planned Enhancements

In its current state, Nickle is a quite useful tool. A number of enhancements to Nickle are planned to increase that usefulness, by making Nickle easier to use and harder to make mistakes in.

First and foremost, while Nickle's current full static type system is a marked improvement over dynamic typing alone, it is not the end of the road. Explicitly stating the types of names is tedious and error-prone, which may partially explain the lack

of strong static typing in most recent scripting language designs. There are two possible improvements over explicit static typing which might be adapted to Nickle with good effect.

Parametric polymorphism, as found in ML and Generic Java (GJ) [BOSW98] (and its close conceptual cousin type templates, as found in C++) tries to capture the idea that an expression that is independent of the details of its input and output types should not be tied to those details. Parametric polymorphism avoids much "copy-and-paste" coding while still retaining the benefits of full static typing. For example, linked list code which is independent of the type of list elements need not be repeated for each possible structure type, a savings in code size and in the potential for error. Since polymorphic typing is fully safe if properly designed and implemented, the downside is minimal: a slight decrease in the expressiveness of the language due to extra constraints on the code.

ML goes a step further, and adds automatic static type inference to the language. Instead of having to explicitly declare all types, undeclared types are inferred from context, and the resulting type assignments are checked for consistency. Experience has shown that polymorphic type inference has much of the flexibility of dynamic typing while still retaining most of the safety of static typing. For a language like Nickle which is to be used as a calculator and for algorithmic prototyping, this seems like an especially ideal combination.

The Nickle implementation is currently several times slower than equivalent C code. This is ameliorated somewhat by the fact that its primitive operations on numbers are well-tuned and exceptionally fast, and by the ability to implement much more sophisticated algorithms in Nickle in equivalent time and code. Nonetheless, there are applications where better performance would be desirable. The obvious approaches of optimization of byte code or compilation to native code and optimization thereof should be explored. There is no reason in principle why Nickle programs should be dramatically slower than C programs, although there is some overhead inherent in the language.

An interesting tack in this direction would be to produce a Nickle compiler to byte code for the Java Virtual Machine (JVM). Running Nickle code under the JVM would bring a couple of important advantages. First, portability to non-UNIX environments would be greatly enhanced. Second, significant performance improvements might

be available “for free”² as a result of the JVM runtime native-code compilation and dynamic code optimization commonly available in many implementations. On the downside, the mismatch between Java’s object-oriented model and Nickle’s imperative-functional model might make things difficult, and the filesystem-based module system of the typical JVM implementation is ill-suited to Nickle’s requirement for separating files from namespaces.

It would be nice if Nickle supported a wider range of built-in types, constants, and operators. The limited range of numeric types is particularly problematic: obvious candidates include the natural numbers, finite fields (particularly $\text{GF}(2^{32})$), and various extensions of the reals, particularly complex numbers. Better semantics and more powerful operators for dealing with vectors, matrices and tensors would also be a plus.

The natural numbers have been in and out of the language at various times in its history. They are currently out, to simplify the language, but they are being reconsidered since their inclusion would close the one known hole in the static type system of the language.³ C-style 32-bit integers were supported at one time, but were deemed to be too much of a special case to expose to the user. Consideration is currently being given to adding general finite field constructors and operators, but there are notable complications.

The lack of complex numbers is slightly embarrassing, but ideas about regularizing them and implementing them in a sane and compatible way have so far been hard to come by. Issues such as the representation of complex coefficients are vexing: for instance, are complex integers desirable? In addition, it is not totally obvious that the complex numbers should be preferred to other extensions of the reals, yet including multiple generalizations makes it difficult to find a suitable static type lattice and correspondingly to choose runtime promotions.

In addition to numeric types, Nickle could arguably use a larger range of modern structured datatypes, such as lists, sets, and curried functions. There are several reasons why we have not provided these to date. Notably, until a polymorphic type system is implemented, static typing issues are difficult to

resolve reasonably. In addition, it is problematic to provide a built-in implementation of datatypes which might be sensibly implemented in multiple ways having different properties. Sets are a particularly good example of this: Pascal’s bitsets have very different runtime properties from sets of integers represented as search tries, and both representations are difficult to generalize to sets of values, such as structures, which have no natural inherent ordering. It is questionable whether the language should make choices for the user about representation issues, and so far it has been shied away from.

While Nickle’s supporting libraries are already well along, further work needs to be done here. The floating-point math support needs to be validated and extended: it would be especially nice to make it compliant with the rounding and precision rules of the IEEE floating point arithmetic standards [IEE85, IEE87]. The string support is extremely rudimentary. Built-in support for array slicing, array comprehensions, and similar features would occasionally be useful. Some implementations of standard Abstract Data Types (ADTs) such as priority queues might occasionally ease algorithm development.

Support for built-in operators on vectors and arrays of numbers would be a real plus, and has no obvious downside except a slight increase in language complexity. The main reason for their lack of current inclusion is the lack of a need for them in the authors’ work.

It would be nice to add some sort of support for workspaces or the like to Nickle, to aid scratch-pad calculation. A save command would be a good start, but in principle Nickle could allow capturing a true continuation and saving it to a file, which would lead to a nice implementation of both workspaces and checkpointing.

3.6 Omitted Features

Perhaps as important as what to include is what not to include. For example, Nickle contains no Object-Oriented Programming (OOP) features. Class-based OOP in the style of C++ and Java is well understood and accepted. However, the combination of this style of OOP with some of the current and proposed features of Nickle, especially polymorphic type inference and first-class functions and continuations, verges on open research questions. In addition, the most useful domains of application of OOP, such as large-scale programming, easy reuse

²As in beer.

³Nickle’s exponentiation operator `**` with an integer base has an integer result when the exponent is a natural number, but a rational result with negative integer argument. This proves to be quite problematic in balancing usability and static typing, and static typing currently loses.

of opaque constructs, and graphics and window system programming, are outside the intended scope of Nickle's applicability.

Other features deliberately omitted from Nickle include:

- Support for graphics and GUI implementation. The impact of this support on the portability and simplicity of the Nickle implementation is potentially large, and the perceived benefit for expected Nickle applications is presently small. Nonetheless, this decision may be revisited in the future.
- Language-level support for interfacing with native code. In actuality, as discussed below, it is quite easy to integrate native code with Nickle programs, but only by integrating the native code into the Nickle implementation.
- Ad-hoc polymorphism, operator overloading, user-defined operators, and related features. This is driven by concern about the complexity and difficulty of implementation of these features, and their perceived negative impact on readability and portability of code. The general philosophy of Nickle is that anything important enough to require these features is important enough to embed in the language definition.

4 History

Nickle began life around 1985 at Reed College as *ec*, a compiler written by Packard for translating arbitrary-precision arithmetic into high-performance interpreted byte code. In addition, around that time, both authors were experimenting with the design and implementation of Kalypso, an interpreter (and later a compiler) for a purely functional dialect of LISP. Inevitably, the desire for LISP-like numerical expressions with C-like syntax led to the construction of *ic*, an "interpreted C" which also incorporated concepts borrowed from earlier work by Packard and others at Tektronix, Inc. on incremental C and Pascal compilation.

The original *ic* was a pure tree-walking interpreter with arbitrary-precision integer and rational datatypes allocated and destroyed statically. This allowed the memory management issues to be finessed. Incremental compilation was one of the first enhancements, and was accompanied by new datatypes which necessitated a reference-counted

memory management scheme with a custom storage allocator.

By about 1993, accumulated incremental changes prompted a complete reworking of the *ic* implementation. The reference-counted storage management was replaced by a tracing mark-sweep collector (borrowed from another of the authors' projects, a functional LISP subset implementation known as Kalypso), first-class functions were added, and the syntax and semantics of the language were revised somewhat. The resulting language was known as Nick. Later additions included first-class continuations, threads, and, by about 1996, namespaces.

In the last 6 months, the static type system has finally been implemented, the platform-native floating point representation has been supplanted by a platform-independent arbitrary-precision implementation,⁴ many builtins have been added, the disjoint union type has been added, the user interface has been improved (including support for GNU *readline*), the documentation has been largely completed, the examples have been collected and regularized, a multitude of bugs and misfeatures have been repaired, and other improvements too numerous to list have been made. The result is Nickle as it exists today.

A few months before the public source release, only minor changes were planned, except for the implementation of polymorphic type inference. As the release was finalized, it became clear that the polymorphic type inference system would have to wait: the features described in the previous paragraph became clear priorities and absorbed all of the available time of both authors. The lesson here is clear and, in retrospect, obvious: the first 90% of Nickle development took 15 years, and the remaining 90% took the last three months. The result appears, so far, to be worth the work: Nickle has never been faster, more stable, or more pleasant to use.

The first public source release of Nickle was in mid-April of 2001. As of the first week since it was announced on freshmeat.net, about 100 copies have been downloaded. So far, there have been no bug reports, and contributions have already been made to the project by users who helped with creating alternative binary packages for the distribution.

Nickle was designed to be highly portable within the UNIX environment: so far, that goal appears to

⁴Several potential floating point representations were considered. In particular, interval arithmetic [Kearney96], while in some ways preferable, was not chosen due to performance concerns.

have been met. Prerelease, it was compiled on a variety of UNIXes with no problems. GNU `autoconf` was invaluable here: while difficult to use, it does its intended job admirably.

5 The Nickle Implementation

Nickle's current implementation consists of about 25,000 lines of C code in about 45 files, together with about 1000 lines of builtins written in Nickle itself. Great attention has been paid to modularity in the implementation: the current structure is the result of literally years of refactoring and reorganization effort. The rough breakdown of the implementation is as follows:

- Builtin datatypes: approximately 20 files, with one `.c` file and one `.h` file per type.
- Memory management: 6 files
- Internal ADT implementations: about 8 files
- Execution infrastructure: about 8 files

The remainder consists of miscellaneous support routines.

As noted above, the implementation uses a highly stereotyped interface to the garbage collector, which allows the C code to easily allocate and reference storage in the Nickle heap. Since this strategy was perfected, memory reference errors in the C infrastructure of the implementation have become extremely rare. Of course, the increasing maturity of the implementation is also a factor here.

Because of the ease of memory management and the extreme modularity, adding new C code to the Nickle implementation is quite easy, even for someone not overly familiar with the internals. As mentioned above, this is one reason why the lack of a native-code interface as part of the Nickle programming language is regarded as unobjectionable. Massey has added C builtins to the implementation on a couple of occasions, and has found the overhead due to learning curve and extra code requirements to be on the order of a few hours. It is not clear that this could be improved with a JNI-style builtin native interface. Portability is an issue, however: the integrated C code should be able to run on arbitrary UNIX (at least) platforms.

6 Experience With Nickle

The various incarnations of Nickle have been used for a range of tasks. First and foremost, Nickle is the calculator program of choice: it is an altogether superior⁵ replacement for UNIX `bc`, `dc`, `expr`, and the like.

Nickle is also a very nice general purpose programming language, especially for numerical work. Nickle programming projects distributed with the reference implementation include

- The Cribbage scoring implementation mentioned above.
- A DSP filter design package.
- Sample data generation for DSP verification.
- A full RSA implementation, including Miller-Rabin probabilistic prime number key generation.
- An implementation (now converted from Nickle to C for use inside Nickle) of Weber's accelerated GCD algorithm.
- A port of the C reference code for the Rijndael encryption algorithm.

In addition, numerous other projects have been assisted by Nickle, including

- Graphics chip clock calculation, and XFree86 "mode line" calculation.
- Probability calculations for Collectible Card Games.
- Course grading.

As noted above, the performance of Nickle is not spectacular, but is adequate for the tasks for which it is intended. For example, the Miller-Rabin implementation typically spends 5–15 seconds generating a 512-bit probabilistic prime on a 700MHz Athlon with adequate memory. This is about a factor of 5 slower than the C-based probabilistic prime generator of OpenSSH. As another example, the Nickle implementation of the Weber GCD code mentioned above is typically 10 times slower on a given input than the C implementation. On the other hand, the Nickle implementation was *much* easier to develop.

⁵Nickle's overhead is considerably larger than those of the listed programs. In practice, this is not a noticeable problem on modern UNIX platforms.

The end result of this experience has been that Nickle has become part of the authors' standard toolkit. It has reached its design goals: the current version is simple to use, extend, and modify.

7 Related Work

The design ideas behind Nickle have been drawn from a number of language implementations, as mentioned above. Relevant languages include C, C++, Icon, Java, ML, Modula-3, Perl, Python, Scheme, UNIX `sed`, AWK, `bc`, `dc` and `expr`, and a host of others. A detailed comparison with each of these language is precluded by space considerations, but some important considerations and principles emerge.

7.1 What Nickle Is Not

First, Nickle is not a text-processing language. While it does include some rudimentary support for strings, and support for file I/O and formatting comparable to (and modeled after) UNIX `stdio`, it has no native support for such niceties as regular-expression-based pattern matching, implicit stream processing, textual variable substitutions, text editing, etc. While the authors have considered the problem of designing a modern text processing language, it would not look a great deal like Nickle; in addition, it is not clear that there is a niche for yet another text processing language given the popularity of many existing candidates.

Second, Nickle is not a language for building large applications. While it does have some support for syntax-level modularity, the implementation is currently rather dependent on whole-program compilation. In addition, the exclusion of OOP and GUI features, as well as the relative inefficiency of the current implementation, augers ill for Nickle's acceptance as a replacement for Ada.

Third, Nickle is not a symbolic algebra package. Its domain is strictly numeric. While a great deal of the Nickle feature set might be useful in a symbolic algebra package, constructing such a thing is probably beyond the purview of a two-person team inexperienced in such matters, and certainly would vastly exceed the current 25K lines of code.

7.2 Comparison With Other Languages

Given the design goal for Nickle—a language for desk calculation and prototyping of numerical and

semi-numerical algorithms—it is constructive to compare its feature set and implementation properties with those of a few of the languages listed above.

First and foremost, unlike all of the languages listed above except certain Scheme implementations, Nickle supports a wide variety of exact or highly precise numeric types, organized in a sensible fashion and properly checked statically and dynamically. The true power of Nickle is not apparent until one tries to add up 10,000 probabilities, and finds that in Nickle they sum to 1; not 0.998 or 1.02, but just plain 1. The ability to select the mantissa precision for floating point computation is similarly useful: it is notable that Nickle is quite useably fast with the default mantissa precision of 256 bits.

A more fair comparison is with R5RS or later Scheme implementations supporting the full numeric model. Such an implementation provides a quite usable calculator and programming language, comparable in some ways to Nickle. The principal differences here include the C-like syntax (indeed, any syntax at all), the static type system, namespaces, structures, etc., all of which ease the sort of programming at which Nickle is aimed. (The lack of built-in list support is a missed feature, as noted above.)

ML has the potential to do much of what Nickle does. Its support for functional programming is obviously far superior to Nickle's, and its syntax, type system, and the like are comparable. The learning curve for ML tends to be fairly steep by most accounts. The experience of the authors is that C programmers pick up Nickle immediately, not just because of the C-like syntax, but also because of the first-class support for imperative programming: Nickle does not try to change one's programming paradigm. Of course, the support for numeric types in Nickle is also superior to that of any ML implementation of which the authors are aware.

Some interest has been expressed in the relationship between Nickle and Perl. First and foremost, as noted above, Perl, `sed` and AWK are aimed primarily at text processing, and are well-suited to this sort of task. The support for numeric programming in Perl is limited: until recently, the only numeric representation supported was IEEE floating point numbers. In addition, the complex syntax and semantics of the language tends to make for a steep learning curve even for experienced programmers [Sch93]. Finally, Perl's complicated system of types and values is somewhat error-prone [McC01] and contains

Table 1: Benchmark execution time in seconds.

	bc	Nickle	GMP
ifact	67.6	5.7	3.4
rfact	67.8	6.0	3.3
choose	130.	6.3	1.8
comp	31.7	9.6	2.6

little support for static typing.

7.3 Performance

Nickle's performance appears to be around 5 times slower than equivalent C code using the GNU GMP multiple-precision library, and quite a bit faster than GNU bc. Some simple benchmarks were run to compare the performance of Nickle 1.99.3, GNU bc 1.05, and C using GNU GMP 2.0. Four benchmarks were utilized: `rfact` computes 20000! using the obvious recursive implementation, `ifact` computes 20000! iteratively, `choose` computes $\binom{20000}{5000}$ (using `ifact` in the C and bc versions), and `comp` applies the Miller-Rabin test to the prime number 31957 for every possible base from 1 to 31956. (The source of all of these benchmarks is available with the Nickle distribution.)

Table 1 shows Nickle execution times on an Athlon 700 with 256MB of RAM running Linux kernel 2.4.1 in single-user mode. All times are the minimum of 5 insignificantly different consecutive runs. (Nickle's built-in `!` operator, while more convenient, produced similar timings to the hand-coded versions.) Nickle and GMP spent about 50% of total time on the factorial benchmarks generating and printing the decimal result (since there appears to be no easy way to inhibit this behavior in bc). The runtimes for these benchmarks are thus somewhat inflated. In general, the performance results are positive: the small performance hit over C code is more than made up for in ease of use.

8 Lessons Learned

Certainly, a number of pragmatic lessons about language design and implementation have emerged over the years of Nickle development. It turns out, for example, to be difficult to give an LALR(1) grammar for such a strong superset of C. Garbage collection turns out to be a huge win over the alternatives: in practice the authors have never observed a problem related to collector performance, and the ease of implementation and the quality of the user experience

have been tremendously improved. The principle of least surprise has proven a good guiding principle for the design: the authors as well as novice users seem to be able to use Nickle without deep thought or constant reference to the documentation.

It was also interesting to observe how two people with similar backgrounds and tendencies can have quite different opinions about even broad details of language design. While the authors always largely agreed on where they were going, there was much involved discussion about the best way of getting there.

In particular, the influence of other languages on the Nickle design was complex and varied: both authors learned a lot about a variety of language options and about how to keep a clear head when evaluating and implementing them. Corner cases in existing language features proved to be problematic: Nickle tended to adopt in a piecemeal fashion features that other languages were designed around, and understanding the best methods of fitting these features in usually required a significant effort.

In preparing the initial draft of this paper, the authors wrote:

The degree of meticulousness [involved in finalizing Nickle] is admittedly unusual in a public utility-belt programming language release. However, it should be understood that the authors have refrained from a public release over a 15-year period precisely to reach this level of quality while there was still room to experiment. Following a successful public release, it will become much harder to make major specification or implementation changes. This drives a desire to get it largely right the first time.

As public release drew near, it became apparent that a number of significant last-minute changes to the language and the implementation were not only desirable but necessary. To a large degree, however, these changes were intended to articulate the goal quoted above: to get the language as "right" as possible before the first public release.

9 Conclusion

Nickle has been an interesting and quite successful experiment in utility-belt programming language

design and implementation. It has increased the authors' understanding of various programming language options, proved out some of their opinions, and been instrumental in getting some of their other work done. We hope it will be useful and interesting for the computing public as well.

Acknowledgments

Thanks to the authors' various employers and institutions over the past 15 years for their tolerance and even encouragement of an effort devoted to a marginally work-related endeavor. Thanks to the Usenix referees for their insightful comments on the paper, and to Clem Cole in particular for his careful (and patient) reading and many helpful suggestions that vastly improved the presentation. Thanks to the pioneers of programming language development, who made the implementation of Nickle both possible and desirable. And a big thanks for the invaluable contributions of users of Nickle and its predecessors over the years, including advice, kibitzing, bug reports and moral support.

Availability

Nickle and a variety of supporting materials are freely available in both source and binary forms from the Nickle web site: <http://www.nickle.org>.

References

- [AWK88] A. V. Aho, P. J. Weinberger, and B. W. Kernighan. *The AWK programming language*. Addison-Wesley, 1988.
- [BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Phillip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Conference on Object-Oriented Programming systems, Languages and Applications (OOPSLA '98)*. SIGPLAN, ACM, October 1998.
- [CE92] William Clinger and Jonathan Rees (Editors). Revised⁴ report on the algorithmic language Scheme. Technical Report CIS-TR-91-25, University of Oregon, February 1992.
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
- [IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, 1985. Revised 1990.
- [IEE87] IEEE. *IEEE 854-1987, Standard for Radix-Independent Floating-Point Arithmetic*. IEEE, New York, NY, 1987. Revised 1994.
- [Kea96] R. Baker Kearfott. Interval Computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95-112, 1996.
- [Lut01] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., second edition, 2001.
- [MC78a] Robert Morris and Lorinda Cherry. *BC - An Arbitrary Precision Desk-Calculator Language*. AT&T Bell Laboratories, 1978. Unix Programmer's Manual Volume 2, 7th Edition.
- [MC78b] Robert Morris and Lorinda Cherry. *DC - An Interactive Desk Calculator*. AT&T Bell Laboratories, 1978. Unix Programmer's Manual Volume 2, 7th Edition.
- [McC01] Jamie McCarthy. Sophomore uses list context; cops interrogate. *Slashdot*, March 2001. URL <http://slashdot.org/yro/01/03/13/208259.shtml>, accessed April 18, 2001 03:52 UTC.
- [McM78] Lee E. McMahon. *SED - A Non-interactive Text Editor*. AT&T Bell Laboratories, 1978. Unix Programmer's Manual Volume 2, 7th Edition.
- [MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [Nel91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.
- [Sch93] Randal L. Schwartz. *Learning Perl*. O'Reilly & Associates, Inc., 1993.
- [WCS96] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., second edition, 1996.

Appendix: The Nickle Tour

Material in typewriter font is taken from a Nickle interactive session. Material in *italics* is commentary.

\$ nickle

The obvious calculations work, including fancy operators and arbitrary precision.

```
> 1 + 1
2
> (2 ** 4)!
20922789888000
```

Rationals are represented exactly, but printed in decimal. Integer division with // is different from rational division.

```
> 1 / 3
0.{3}
> . * 3
1
> 1 // 3
0
```

Expected conveniences, like the value . denoting the last value printed, work. Implicit declarations work at top level, as do explicit typed declarations. Using a statement form at top level results in no value being printed.

```
> x = .
0
> int y = x;
```

C-style statements can be typed at the command line. The + prompt denotes an incomplete statement.

```
> for (int i = 1; i <= 9; i += 2)
+   x += i;
> x
25
```

Exact integer square roots will be represented by integers. For irrational roots, a 256 bit floating point representation is used; the printed representation is indistinguishable.

```
> xsqrt = sqrt(x)
5
> sqrt(2)
1.4142135623730
> . * .
2
> sqrt(5)
2.2360679774997
> . * .
4.9999999999999
```

Functions may be typed at the command line. Argument and result types are optional. This function returns nonsense for non-integers.

```
> function sqr(x) {
+   auto s = 0;
+   for (int i = 1;
+       i < 2 * x + 1;
+       i += 2)
+     s += i;
+   return s;
+ }
> sqr(5)
25
> sqr(5.1)
36
```

Functions are first-class: untyped functions can be assigned to statically typed function variables.

```
> int(int) isqr = sqr;
> isqr(5)
25
> isqr(5.1)
-> isqr ((51/10))
Incompatible types 'int', 'rational'
argument 0
```

Operators try to behave properly in as many cases as possible.

```
> 5.1 ** 2
26.01
> -5.1 ** 2
26.01
> -5.1 ** 3
-132.651
> -5.1 ** 3.1
Unhandled exception "invalid_argument"
at /usr/local/share/nickle/math.5c:196
"log: must be positive"
0
(-51/10)
> quit
```

For reasonable sized chunks of code, it is normal to use a separate text file.

```
$ cat > stack.5c
namespace Stack {
    typedef frame;
    typedef struct{
        poly val;
        *frame next;
    } frame;
    public typedef * *frame stack;
    public exception stack_underflow();

    public stack function
    new() {
        return reference(0);
    }

    public void function
    push(stack s, poly xval) {
        *s = reference((frame){
            next = *s,
            val = xval
        });
    }

    public poly function
    pop(stack s) {
        if (*s == 0)
            raise stack_underflow();
        poly xval = (*s)->val;
        *s = (*s)->next;
        return xval;
    }
}
~D
```

Here is the Stack ADT in action.

```
$ nickle
> load "stack.5c"
> print Stack
namespace Stack {
    public typedef **frame stack;
    public stack function new ();
    public void function
        push (stack s, xval);
    public function pop (stack s);
}
> import Stack;
> stack s = new()
&0
> push(s, "x")
> push(s, 3)
```

```
> pop(s)
3
> pop(s)
"x"
```

Uncaught exceptions lead to the debugger.

```
> pop(s)
Unhandled exception "stack_underflow"
at stack.5c:23
- trace
    raise stack_underflow ();
pop (&0)
    pop (s)
- s
&0
- done
> quit
$
```

The Design and Implementation of the NetBSD rc.d system

Luke Mewburn

Wasabi Systems, Inc.

lukem@wasabisystems.com

Abstract

In this paper I cover the design and implementation of the rc.d system start-up mechanism in NetBSD 1.5, which replaced the monolithic */etc/rc* start-up file inherited from 4.4BSD. Topics covered include a history of various UNIX start-up mechanisms (including NetBSD prior to 1.5), design considerations that evolved over six years of discussions, implementation details, an examination of the human issues that occurred during the design and implementation, as well as future directions for the system.

1. Introduction

NetBSD recently converted from the traditional 4.4BSD monolithic */etc/rc* start-up script to an */etc/rc.d* mechanism, where there is a separate script to manage each service or daemon, and these scripts are executed in a specific order at system boot.

This paper covers the motivation, design and implementation of the rc.d system; from the history of what NetBSD had before to the system that NetBSD 1.5 shipped with in December 2000, as well as future directions.

The changes were contentious and generated some of the liveliest discussions about any feature change ever made in NetBSD. Parts of those discussions will be covered to provide insight into some of the design and implementation decisions.

2. History

There is great diversity in the system start-up mechanisms used by various UNIX variants. A few of the more pertinent schemes are detailed below. As NetBSD is derived from 4.4BSD, it follows that a description of the latter's method is relevant. Solaris' start-up method is also detailed, as it is the most common System V UNIX variant.

2.1. 4.4BSD

4.4BSD has a rather simple start-up sequence.

When booting multi-user, the kernel runs *init* (located in */sbin/init*), which spawns a shell (*/bin/sh*) to run */etc/rc*, which contains commands to check the consistency of the file-systems, mount the disks, start up system processes, etc. */etc/rc* invokes */etc/netstart* to configure the network and any associated services, and */etc/rc.local* (if it exists) for locally added services. After */etc/rc* has successfully completed, *init* forks a copy of itself for each terminal in */etc/ttys*, usually running */usr/libexec/getty* on them. [1]

Administrative configuration of system services is controlled by editing the scripts (*/etc/rc*, */etc/rc.local*, */etc/netstart*). In some instances, only shell variables need to be changed, in others commands are added, changed, or removed. [2]

4.4BSD has no specific shut down procedure. After *init* receives a SIGTERM signal it sends a SIGHUP signal to each process with a controlling terminal, which the process was expected to catch and handle appropriately. Ten seconds later, this is repeated with SIGTERM instead of the SIGHUP, and another ten seconds after that SIGKILL is sent. After all processes have exited or when thirty seconds had elapsed, *init* then drops to single user mode, reboots, or shuts down, as appropriate.

2.2. Solaris 7

Solaris is the most common System V variant, and serves as a good reference implementation of the System V *init.d* mechanism, as implemented by System V Release 4 (SVR4).

When running, the system can be in one of eight distinct run levels [3], which are distinct states in which selected groups of processes may run. The run level may be changed at any time by a privileged user running the

`init` with the run level as the argument, and the run level may be determined at any time with the “`who -r`” command.

When the system is booted, the kernel runs `init` (located in `/sbin/init`), whose purpose is to spawn processes defined in `/etc/inittab` [4]. For each configuration line in `/etc/inittab` that has a run level field (‘`rlevel`’) which matches the current run level, `init` starts the process defined on that line as per the given ‘`action`’ field. The different run levels are:

0	Shut down the operating system so that it's safe to turn off the power.
s or S	Single user mode, with all file systems mounted.
1	Single user mode, with all file systems mounted and user logins allowed.
2	Multi user mode, with all services running except NFS server daemons.
3	Multi-user mode with all services running. This is usually the default.
4	Currently unused.
5	Shut down the system and attempt to turn off the power.
6	Shut down the system to level 0, and reboot.

For a given run level *X*, an shell script `/sbin/rcX` exists to control the run level change, and `/etc/rcX.d` contains scripts to be executed at the change. `/sbin/rcX` stops the services in the files matching `/etc/rcX.d/K*` in lexicographical order, and then starts the services matching `/etc/rcX.d/S*` in order.

To add a new service *foo* requires adding `/etc/rcX.d/S*foo` in the appropriate run level to start the service, and then `/etc/rcY.d/K*foo` in all the other run levels *Y* where the service is not to be run. Usually these files are actually links to the appropriate script in `/etc/init.d` which implements the start up and shut down procedures for a given service.

To disable or remove a service *foo*, any files matching `/etc/rc?.d/[KS]*foo` need to be removed.

2.3. NetBSD prior to 1.5

Prior to the release of NetBSD 1.3, NetBSD's start-up mechanism was similar to 4.4BSD's, with relatively minor changes, as described below.

2.3.1. NetBSD 1.3

In NetBSD 1.3 (released in January 1998), two major user-visible additions were made to the start-up system; `/etc/rc.conf` and `/etc/rc.lkm`.

`/etc/rc.conf` contains variables to control which services are started by `/etc/rc` and `/etc/netstart`. For each service *foo*, two variables may be provided:

<code>\$foo</code>	Can be “yes” or “no” (or various other boolean equivalents). If set to “yes”, the service or action relating to <i>foo</i> is started.
<code>\$foo_flags</code>	Optional flags to invoke <i>foo</i> with.

The aim of `/etc/rc.conf` was to separate the scripts that start services from the configuration information about the services. This allows updating of the start-up scripts in an operating system upgrade with less chance of losing site-specific configuration.

Similar `/etc/rc.conf` functionality has been implemented in commercial UNIX and BSD derived systems, including current systems such as FreeBSD. By the time this change was considered for NetBSD, it had a reasonable number of users of the prior art to help justify its implementation.

`/etc/rc.lkm` was added to provide control over how loadable kernel modules (LKMs) are loaded at boot time. `/etc/rc.lkm` is invoked at three separate stages during the boot process; before networking is started, before non-critical file systems (i.e., file systems other than `/`, `/usr`, `/var`) are mounted, and after all file-systems are mounted. This complexity is required because an LKM may be located on a local or remote file system. The configuration file `/etc/lkm.conf` controls behavior of `/etc/rc.lkm`.

2.3.2. NetBSD 1.4

In NetBSD 1.4 (released in May 1999), two more additions were made; `/etc/rc.shutdown` and `/etc/rc.wscns`.

`/etc/rc.shutdown` is run at shut down time by `shutdown`. This occurs before the global `SIGHUP` is sent (as described in section 2.1). This is useful because there

are some services that should be shut down in order (e.g., database-using applications before their databases) and some services that require more than SIGHUP for a clean shutdown.

`/etc/rc.wscons` was added to control how the `wscons` console driver was configured at boot time, and to allow manual reconfiguration. `/etc/wscons.conf` controls this behavior.

2.3.3. Summary prior to NetBSD 1.5

At multiuser boot, `init` calls `/etc/rc` to initialize the system. `/etc/rc` calls `/etc/netstart` to setup network services, `/etc/rc.local` for local services, `/etc/rc.lkm` to initialize load-able kernel modules, and `/etc/rc.wscons` to configure the `wscons` console driver. The start-up of services is controlled by variables in `/etc/rc.conf`.

At system shutdown time, `shutdown` calls `/etc/rc.shutdown` to shut down specific services which have to be shut down before the global SIGHUP that `init` sends.

3. Design considerations

Over a six year period, various ideas on how to enhance the start-up system were floated on the public NetBSD mailing lists 'current-users' and 'tech-userlevel', as well as on the NetBSD developer-only mailing list.

There was no consensus on 'One True Design'; there was too much contention for that. What is described below is an amalgamation of what a few developers felt was a reasonable analysis of the problems and feedback as well as the most reasonable solution to support the widest variety of circumstances.

3.1. Problems with the old system

The old system was perceived to suffer from the following problems:

- There was no control over the dependency ordering, except by manually editing `/etc/rc` (and other scripts) and moving parts around.

This caused problems at various times, in situations such as workstations with remotely mounted `/usr` partitions, and these problems weren't completely resolved as was seen by observing various mailing discussions and a flurry of CVS commits to the source tree.

- It was difficult to manually control an individual service after the system booted (e.g., restart `dhcpcd`, shut down a database, etc).

Whilst some people suggested that a system administrator who couldn't manually restart a service was incompetent, this doesn't resolve the issue that typing "`/etc/rc.d/amd restart`" is significantly easier than finding the process identifier of `amd`, killing it, examining `/etc/rc` for the syntax that `amd` is invoked with, searching `/etc/rc.conf` for any site-specific options, and manually typing in the resulting command.

Unfortunately, there was a slight tendency during some of the mailing list discussions to resort to attacks on people's competency in this manner. I consider this a form of computer based intellectual snobbery, and an unreasonable justification for why that person disliked a feature.

- It didn't easily cater for addition of local or third party start-up mechanisms, especially addition into arbitrary points in the boot sequence, including those installed by (semi-)automated procedures such as the NetBSD 'pkg' tools.

3.2. Requirements of the new system

Given the problems in the old system, and observations of what other systems have done, including those described in section 2, the following design considerations were defined.

Some of these considerations were not determined during discussion prior to implementation, but were identified once users were actively using the implementation.

3.2.1. Dependency ordering

Dependency ordering is a strong requirement.

The following dependency ordering requirements were determined:

- Independence from lexicographical ordering of filenames.

Some other systems (e.g., System V `init.d`) use an existing lexicographical ordering of filenames in a given directory, such as `/etc/rc2.d/S69inet` occurring before `/etc/rc2.d/S70uuucp`, but experience has shown that this doesn't necessarily scale cleanly when adding local or third-party services into the

order; often you end up with a lot of convoluted names around 'S99'.

- Ability to insert local or third-party scripts anywhere into the sequence.

Some people proposed running */etc/rc.d/** out of */etc/rc.local*, and retaining the existing */etc/rc* semantics. This doesn't easily cater to a user who requires the ability to insert their own start-up items anywhere in the boot sequence (such as a cable modem authentication daemon required for net-working).

- Not bloating */bin* and */sbin* on machines with small root (*/*) file-systems. The use of tools from */usr/bin* has to be avoided because */usr* might not be available early in the boot sequence.
- Use a dynamic dependency ordering.

A lot of debate occurred regarding whether the dependency ordering is predetermined (e.g., by creating links to filenames or building a configuration file), or dynamically generated.

A predetermined order may be more obvious to determine the order (using 'ls' or examining the configuration file instead of invoking a special command), but it can be difficult to add a service in at a given point on a system because generally ordering is not based on services provided.

A dynamic order may slow down boot slightly, but provides the flexibility of specifying start-up order in terms of dependencies.

For example, if service *C* depends on *B* which depends on *A*, and I have a new service *D* to install that must start after *A* and before *C* then I want to specify it in these terms, without having to worry about whether it starts before *B*, after *B*, or simultaneously with *B*.

There was some discussion about various methods in which to determine the dynamic ordering:

- Using *make* and a *Makefile*.
- Using *tsort*, *awk*, and a few shell commands
- Providing a dedicated ordering tool which parsed the scripts for command directives in special comments to determine the order. If a

script did not have a directive, it would be ordered last.

After various discussions and implementation tests, it was decided that a dedicated dynamic ordering tool, *rcorder* (see section 4.2.6), was the most appropriate mechanism; using *make* or *tsort* and *awk* would require moving those programs to */bin* ('bloating' the root file-system for machines with limited resources), and a dedicated tool could provide better feedback in certain error situations.

3.2.2. Manipulation of individual services

Most people seem to agree that the ability to manipulate an individual service (via a script) is one of the benefits of the System V *init.d* start-up mechanism. Having a script that allows direct starting, stopping, and restarting of a service, as well as other per-service options like 'reloading configuration files', significantly reduces system administrator overheads.

Having the same script be used by the start-up sequence is also highly desirable, as opposed to using a monolithic */etc/rc* for booting and separate */etc/rc.d* scripts for manual control (which had been suggested).

It is interesting to note that some System V *init.d* implementations often start multiple services in the one file, which defeats the purpose of providing per-service control files. An example is Solaris' */etc/init.d/inetsvc*, which configures network interfaces, starts *named* and starts *inetd*.

3.2.3. Support third-party scripts

An important requirement is the ability to support third-party scripts, especially by allowing them to be inserted at any place in the boot sequence order.

The current system does support third-party scripts if they are installed into */etc/rc.d*. There has been discussion about allowing for different directories to be used for local and third-party scripts, in order to provide a separate 'name-space' to prevent possible conflicts with a local script and a future base system script, but so far none of the suggestions has been considered sufficiently complete to provide in the default system. This, however, does not prevent a site from implementing their own method.

3.2.4. Maintain */etc/rc.conf*

/etc/rc.conf was introduced in NetBSD 1.3, and most users seem fairly happy with the concept.

One of the concerns about a traditional System V *init.d* style mechanism is that the control of service start-up is managed by the existing of a link (or symbolic link) from */etc/rc2.d/S69inet* to */etc/init.d/inetinit*, which is difficult to manage in a traditional configuration change management environment (such as RCS). Similar concerns exist regarding the suggestion of using mode bits on files in */etc/rc.d* to control start-up.

/etc/rc.conf was further enhanced as described in section 3.3.

3.2.5. Promote code re-use

Traditional System V *init.d* implementations do not appear to re-use any code between scripts. From experience, maintaining local scripts in a traditional *init.d* environment is a maintenance nightmare. We achieved code re-use with common functions in */etc/rc.subr* which results in the average */etc/rc.d* script being a small (5-10 line) file. There were some concerns raised about using these common functions, but they weren't considered to be serious issues. (We have a C library and common Makefile fragments, so why not common shell functions?)

3.2.6. Service shut down

The ability to shut down certain services at system shutdown time with */etc/rc.shutdown* was a useful feature of the previous system and of other systems, and it makes sense to retain this feature.

In the initial implementation, we reverse the dependency order, and shut down any services which are tagged with a 'shut down' keyword (see section 4.2.6) within the script. We may modify or enhance this behavior if observation of in-field use reveals a more complicated scheme is required.

3.2.7. Avoid mandatory run levels

We avoided the use of System V run levels (also known as run states or init states) and */etc/inittab*. This was the result of many discussions about the design, which can be summarized to:

- They're just too contentious; the */etc/inittab* concept had the least number of advocates. Many people expressed the opinion (both during the design phase and post implementation) that they don't

mind the */etc/rc.d* idea but don't think an */etc/inittab*, run-levels or */etc/rcN.d* directories would improve things.

- There doesn't seem to be consistency between what each run-level means on various System V *init.d* implementations, or the exact semantics of what occurs at state change. Thus, using the argument of compatibility for system administrator ease of use isn't as relevant. Some systems (such as HP/UX 10.x) treat these as levels, where a transition from level 4 to level 2 executes the shut down scripts in level 3 and then level 2. Other systems (such as Solaris) treat these as separate run states, where a transition to a level runs all the stop scripts in that level and then all the start scripts. This can be confusing to an administrator, as well as not necessarily providing the optimal behavior.
- We currently support single-user mode (s), multi-user mode (2 or 3, depending on whether NFS serving is configured in */etc/rc.conf*), shutting down the system to single user mode (1), halting the system (0), rebooting the system (6), and powering off the system (5) (with the equivalent Solaris init state in parenthesis).
- If the ability to take the system from a given point in the order to another point in the order, then I feel that most people's requirements for what run-levels are touted to provide would be met. This is currently a work in progress.
- Whilst */etc/inittab* provides for re-spawning of daemons, in practice very few daemons are actually started that way, and it's trivial to implement that feature in a few lines of shell script as a 'wrapper' to the start of the daemon.

3.2.8. Other issues

After various discussions, we settled on the name */etc/rc.d* instead of */etc/init.d*, because the implementation was different enough from the System V */etc/init.d* mechanism that we decided not to confuse people expecting the exact System V semantics. Many system administrators may be used to referring directly to */etc/init.d/foo* or */sbin/init.d/bar* when manipulating a service; a symbolic link from */etc/init.d* or */sbin/init.d* to */etc/rc.d* on their systems could help retain their sanity.

The first implementation of */etc/rc.d* that I released for evaluation supported all three start-up schemes; the original monolithic */etc/rc*, a System V *init.d* (without

run-levels), and the current */etc/rc.d*. These were all built from the same sources, and a command was provided to generate the style that an administrator preferred. After feedback and discussion, this functionality was abandoned, because:

- It is very difficult to support multiple ways of starting the system when users have problems or questions, especially so in a volunteer project.
- Two of the methods (*/etc/rc*, and System V *init.d*) do not have the ability to dynamically order the dependency list. In those situations, an administrator (or automatic application) would have to perform the extra step of 'rebuild order' upon installation.
- The source scripts had various constraints to ensure that they could work as part of */etc/rc* as well as acting as a stand-alone script in */etc/rc.d* or */etc/init.d*.

As architects of the NetBSD operating system, we have the responsibility to provide useful solutions to problems. In general, those solutions should be as flexible as possible, without introducing *unnecessary* flexibility, which will only cause confusion. Therefore, the alternative mechanisms were dropped.

That said, the current system is flexible enough that if a site decided to use a System V *init.d* approach, it is fairly trivial to populate */etc/rcN.d* with a symbolic link farm to files in */etc/rc.d* (using *rcorder* to build the dependency list), and modify */etc/rc* to run the scripts in */etc/rcN.d/* in lexicographical order, or to even implement a System V */etc/inittab* and run states.

Unfortunately, there is no easy solution for people who want to retain */etc/rc*. However, as NetBSD is an Open Source project and allows for public access to the CVS source code repository (via anonymous CVS as well as via a WWW front-end [6]), nothing prevents users from reverting to the old style */etc/rc*.

It is interesting that the people who argued the most to retain */etc/rc* are probably those who are skilled enough to maintain this, and during the various discussions some even offered (some might say "threatened") to maintain their own copy of */etc/rc* in their own public CVS server for those who wished to retain this functionality. Interestingly, over a year has passed since the implementation of this work and there is no evidence that any */etc/rc* splinter work has actually occurred.

3.3. Configuration improvements

The */etc/rc.conf* mechanism was enhanced in two ways:

- The default configuration settings were moved from */etc/rc.conf* to */etc/defaults/rc.conf*, and */etc/rc.conf* sources the former. Site specific configuration overrides are placed in */etc/rc.conf*. This enables easier upgrades (both manual and automatic) of the default settings in */etc/defaults/rc.conf* for new or changed services.

There was debate about this change, but a significant majority of users agreed with the change. Also, FreeBSD had made a similar change some time before, with a similar debate and outcome, and subsequent upgrade benefits observed which helped the case supporting the change.

- An optional per-service configuration file in */etc/rc.conf.d/SERVICE* was added. This configuration file (if it exists) is read after */etc/rc.conf*, to allow per-service overrides. This optional functionality was added to allow automated third-party installation mechanisms to easily add configuration data.

Migrating entirely away from */etc/rc.conf* to a multitude of */etc/rc.conf.d/SERVICE* files was considered, but no consensus was reached, and after a local trial, we decided that providing for the latter but retaining the former satisfies proponents of either side.

Recently (post NetBSD 1.5), a */sbin/chkconfig* command has been added (similar to the equivalent command in IRIX) to manage */etc/rc.conf.d* by displaying a setting or changing its value.

Thus, the order that configuration information for a given service *foo* is read in is as follows:

- *foo* sources */etc/rc.conf*.
- */etc/rc.conf* sources in */etc/defaults/rc.conf* (if it exists), and machine specific overrides of the defaults are added at the end of */etc/rc.conf*.
- A per-service configuration file in */etc/rc.conf.d/foo* (if it exists) will be loaded. This allows for automated maintenance of */etc/rc.conf.d* configuration files, whilst retaining the popular */etc/rc.conf* semantics.

4. Implementation & aftermath

The system was implemented as described above in the design section, although the design was slightly fluid and did change as feedback was incorporated.

There are two elements to the post-implementation analysis; the human issues, and the technical details.

4.1. The human issues

There was a lot of feedback, debate, angst, flames, and hate-mail. The change has been one of the most contentious in the history of the project.

The first commits to the source code repository were made with the intention of providing a mostly complete implementation which was to be incrementally improved over a few months before the release of NetBSD 1.5.

Unfortunately, we made one of our largest implementation mistakes at this point; we didn't warn the user-base that this was our intention, and the commits were seen as a 'stealth attack'. This was partly because we felt that there had been enough debate and announcing our intentions would have delayed the project another few months for a rehash of the same debate (which had been going on for five years at that point).

After the initial implementation, various technical and 'religious' complaints were raised about the system. A summary of these is:

- *"The use of 'magic' functions [from /etc/rc.subr] is bad."*

It was felt that the code re-use that `/etc/rc.subr` promotes was sufficiently worthy to justify its continued use, as described in section 3.2.5.

- *"Switching from /etc/rc is not the BSD way, ..."*

This particular objection was expected; it's a religious argument and the change was bound to annoy a certain section of the community.

Robert Elz, a long time user and contributor to BSD, had a good point to make about 'the BSD way': *"[the BSD way is to] find something that looks to be better (in the opinion of the small group deciding such things), implement it, and ship it."*[5]

In this case, the 'small group' was the NetBSD core team, who voted in unanimous agreement for the

work, with the proviso that it would be tweaked and improved as necessary, which is what occurred.

- *"Why wasn't a System V `init.d` implemented?"*

This was covered in section 3.2.7.

Because some of the detractors were quite vocal in the complaints, there was a perception for a time that the work was against a majority decision. This was far from the truth; many users and developers had become jaded with the discussion over the years and did not bother to argue in support of the change, since they agreed with it in principle, if not in implementation particulars. This was borne out by the level of support for the change in the time since implementation.

4.2. The technical details

The `rc.d` system comprises of the following components:

<code>/etc/rc</code>	System start-up script.
<code>/etc/rc.shutdown</code>	System shutdown script.
<code>/etc/rc.d/*</code>	Individual start-up scripts.
<code>/etc/rc.subr</code>	Common shell code used by various scripts.
<code>/etc/defaults/rc.conf</code>	Default system configuration.
<code>/etc/rc.conf</code>	System configuration file.
<code>/etc/rc.conf.d/*</code>	Per service configuration file.

4.2.1. `/etc/rc`

On system start-up, `/etc/rc` is executed by `init`.

`/etc/rc` then calls `rcorder` to order the scripts in `/etc/rc.d` that do not have a 'nostart' `rcorder` keyword to obtain a dependency list of script names. `/etc/rc` then invokes each script in turn with the argument of 'start' to start the service.

The purpose of the 'nostart' support is to allow (primarily third-party) scripts which are only to be manipulated manually (and not started automatically) to be installed into `/etc/rc.d`. No scripts in the standard NetBSD distribution use this feature as yet.

4.2.2. */etc/rc.shutdown*

At system shutdown, */etc/rc.shutdown* is executed by shutdown, halt, reboot and poweroff do not call this script.

/etc/rc.shutdown then calls *rcorder* to order the scripts in */etc/rc.d* that have a 'shutdown' *rcorder* keyword to obtain a dependency list of script names. This dependency list is then reversed, and */etc/rc.shutdown* then invokes each script in turn with the argument of 'stop' to stop the service.

The rationale for this is that only a few services (such as databases) actually require a shutdown mechanism more complicated than the SIGHUP sent by */sbin/init* at shutdown time. Also, having every script perform 'stop' slows down system shutdown as well as causing problems in other areas (such as cleanly un-mounting a 'busy' NFS mount once the networking services have been stopped).

4.2.3. */etc/rc.d/** scripts

The scripts in */etc/rc.d* are invoked by */etc/rc* (with an argument of 'start') and */etc/rc.shutdown* (with an argument of 'stop') in the order specified by *rcorder* to start and stop (respectively) a given service.

The */etc/rc.d* scripts can be invoked manually by a system administrator to manipulate a given service (such as starting, reloading configuration, stopping, etc.)

Each script should support the following (mutually exclusive) arguments:

start	Start the service. This should check that the service is to be started as controlled by <i>/etc/rc.conf</i> . Also checks if the service is already running and refuses to start if it is. This latter check is not performed by standard NetBSD scripts if the system is starting directly to multi-user mode, to speed up the boot process.
stop	Stop the service if <i>/etc/rc.conf</i> specifies that it should have been started. This should check that the service is running and complain if it is not.

Other arguments which are supported in the standard NetBSD */etc/rc.d* scripts include:

restart	Effectively perform 'stop' then 'start'.
status	If the script starts a process (rather than performing a one-off operation), show the status of the process. Otherwise, it's not necessary to support this argument. Defaults to displaying the process ID of the service (if running).
rcvar	Display which <i>/etc/rc.conf</i> variables are used to control the start-up of the service (if any).

If the argument is prefixed by 'force', then tell the script to as if the */etc/rc.conf* variable which controls that service's start-up has been set to 'yes'. This allows a system administrator to manually control a service disabled by */etc/rc.conf* without editing the latter to enable it. This does not skip the check which determines if the service is already running.

Other arguments for manual use by a system administrator (such as 'reload', etc) can be added on a per service basis. For example, */etc/rc.d/named* supports 'reload' to reload named's configuration files without interrupting service.

There are some 'placeholder' services which can be required by a service to ensure that it is started before or after certain operations have been performed. These scripts generally have a name that is all upper case, and in the order found in the default boot sequence are:

NETWORK	Ensure basic network services are running, including general network configuration (<i>network</i>), and <i>dhclient</i> .
SERVERS	Ensure basic services (such as <i>NETWORK</i> , <i>ppp</i> , <i>syslogd</i> , and <i>kdc</i>) exist for services that start early (such as <i>named</i>), because they're required by <i>DAEMON</i> below.
DAEMON	Before all general purpose daemons such as <i>dhcpcd</i> , <i>lpd</i> , and <i>ntpd</i> .
LOGIN	Before user login services (<i>inetd</i> , <i>telnetd</i> , <i>rshd</i> , <i>sshd</i> , and <i>xdm</i>), as well as before services which might run commands as users (<i>cron</i> , <i>postfix</i> , and <i>sendmail</i>).

4.2.4. */etc/defaults/rc.conf*, */etc/rc.conf*, */etc/rc.conf.d/**

/etc/defaults/rc.conf contains the default settings for the standard system services, and is provided to facilitate easier system upgrades. End users should not edit this file.

/etc/rc.conf is the primary system start-up configuration file. It reads in */etc/defaults/rc.conf* (if it exists), and the end-user puts site-local overrides of these settings at the end of the */etc/rc.conf*. This makes it more obvious to between what is a system default and what is a site-local change, and provides similar functionality to FreeBSD's */etc/defaults* mechanism.

For a given service *foo*, it is possible to have a per-service configuration file in */etc/rc.conf.d/foo*, which is read after */etc/rc.conf*. This was provided to allow third-party installation tools to install a default configuration without requiring them to in-line edit */etc/rc.conf*.

Example */etc/rc.conf* entries for *dhclient* are:

```
dhclient=YES
dhclient_flags="-q tlp0"
```

To ensure that the system doesn't start into multi-user mode without the system administrator actually checking the configuration of the system, the variable 'rc_configured' is set to 'no' by default, and must be set to 'yes' by the system administrator. If this is not set, the system will not boot into multi-user mode, and instead remain in single-user mode. The system installation tool 'sysinst' makes this change for you when configuring a newly-installed system, but users performing manual installations or upgrades need to be aware of this.

As */etc/rc.conf* is a shell script, it is possible to put various shell commands into the script to conditionally set flags if necessary. Be aware, however, that if the script exits then any script that sources */etc/rc.conf* (such as the system boot scripts) will exit too. As */etc/rc.conf* may be loaded early in the boot sequence (possibly before */usr* is mounted), not all commands may not be available for use.

4.2.5. */etc/rc.subr*

/etc/rc.subr is a shell script that's sourced by the */etc/rc.d* scripts. It contains 'helper' shell functions for commonly used operations:

- *checkyesno var*

Determine if the given variable *var* is set to 'yes' or 'no' (or a variant), and return with an exit code of 0 if yes, 1 if no.

- *check_pidfile pidfile procname*

Parse the first line of the specified file *pidfile* for a PID, and print the PID if that process is running and matches the given process name *procname*.

- *check_process procname*

Print a list of PIDs that match the given process name *procname*.

- *load_rc_config command*

Load in the *rc.conf* configuration for *command*, first from */etc/rc.conf*, and then from */etc/rc.conf.d/command* (if it exists.)

- *run_rc_command arg*

Implement the 'guts' of an *rc.d* script. This is highly flexible, and supports many different service types. *arg* is argument describing the operation to perform (e.g., 'start' or 'stop'). The behavior of *run_rc_command* is controlled by shell variables defined before invoking the function.

In traditional System V *init.d* systems (e.g., Solaris), each script contains the code to determine if a script should be started or shut down, and often re-implemented the checks for a running process, etc. These scripts become difficult to maintain, and are often 1-2 pages long.

By using the functions in */etc/rc.subr*, the standard NetBSD *rc.d* scripts are quite small in comparison.

For example, the */etc/rc.d/dhclient* script (sans comments which aren't used by *rcorder*) is:

```
#!/bin/sh
#
# PROVIDE: dhclient
# REQUIRE: network mountcritlocal

. /etc/rc.subr

name="dhclient"
rcvar=$name
command="/sbin/${name}"
pidfile="/var/run/${name}.pid"
load_rc_config $name
run_rc_command "$1"
```

It is not mandatory for scripts to use these functions. An ordinary shell script (with the appropriate *rcorder* control comment lines) which supports the arguments 'start' and 'stop' should work at system start-up and shutdown without modification. In order to be consistent with the existing *rc.d* scripts, it may help to also support 'restart', 'status', 'rcvar' (if appropriate), as well as the 'force' prefix.

4.2.6. rcorder

The ordering of the scripts in */etc/rc.d* is performed by *rcorder* (located in */sbin/rcorder*), which prints a dependency ordering of a set of interdependent scripts. *rcorder* reads each script for special comment lines which describe how the script is dependent upon other services, and what services this script provides.

Example *rcorder* comment lines for */etc/rc.d/dhclient* follow:

```
# PROVIDE: dhclient
# REQUIRE: network mountcritlocal
```

In this case, *dhclient* requires the services 'network' (to configure basic network services) and 'mountcritlocal' (to mount critical file-systems required for early in the boot sequence, usually */var*), and provides the service 'dhclient' (which happens to be depended upon by the placeholder script */etc/rc.d/NETWORK*).

It is possible to tag a script with a keyword which can be used to conditionally include or exclude the script from being returned by *rcorder* in the result. */etc/rc* uses this to exclude scripts that have a 'nostart' keyword, and */etc/rc.shutdown* uses this to only include scripts that have a 'shutdown' keyword. For example, as *xm* needs to be shut down cleanly on some platforms, */etc/rc.d/xm* contains:

```
# KEYWORD: shutdown
```

The *rcorder* dependency mechanism enables third-party scripts to be installed into */etc/rc.d* and therefore added into the dependency tree at the appropriate start-up point without difficulty.

5. Future Work

I'd like to implement functionality to allow you to start up (or shut down) services from service *A* to service *B*. This would allow you to start in single user mode, and then start up enough to get the network running, or start all services until just before 'multi user login', or just those between 'network running' to 'database start', etc.

This could be a fairly simple system, and would provide most of the functionality that most people seem to want run states for.

I encourage other systems that are still using a monolithic */etc/rc* and who would like to resolve some of the similar issues NetBSD had, to consider this work. I would like to liaise with the maintainers of those systems to ensure as much code re-use as possible.

6. Conclusion

NetBSD 1.5 has a start-up system which implements useful functionality such the ability to control the dependency ordering of services at system boot and manipulate individual services, as well as retaining useful features of previous releases such as */etc/rc.conf*.

This work was extremely contentious and difficult to implement because of this contentious nature. The implementation phase did provide valuable insight into some of the difficulties involved in the design and development of large open source projects.

In the long run I believe that this work will have benefits for a majority of users, both in day-to-day operation of the system as well as during future upgrades from NetBSD 1.5.

Availability

This work first appeared in NetBSD 1.5, which was released in December, 2000 [7]. The CVSweb interface [6] can be used to browse the work and its CVS history.

Acknowledgements

Many people contributed to the discussions and design of the current system.

However, some people in particular provided some of the important elements: Matthew Green for finishing rcorder and providing the initial attempt at splitting */etc/rc* into */etc/rc.d*, and Perry Metzger for the idea of providing dependencies using a ‘PROVIDE’ and ‘REQUIRE’ mechanism, and for the initial rcorder implementation.

References

- [1] M. K. McKusick, K. Bostic, M. J. Karels, & J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, Reading, MA, 1996.
- [2] M. K. McKusick, K. Bostic, M. J. Karels, & S. J. Leffler, “Installing and Operating 4.4BSD UNIX”, in *4.4BSD System Managers Manual*, pp. 1:44-53, O’Reilly & Associates, Sebastopol, CA, 1994.
- [3] Sun Microsystems, *System Administration Guide, Volume I*, Sun Microsystems, Palo Alto, CA, 1998.
- [4] Sun Microsystems, “init(1M)”, in *man Page(1M) System Administration Commands*, Sun Microsystems, Palo Alto, CA, 1998.
- [5] Robert Elz, in email to tech-userlevel@netbsd.org: <http://mail-index.netbsd.org/tech-userlevel/2000/03/17/0010.html>
- [6] The NetBSD Project, “CVS Repository”, <http://cvsweb.netbsd.org/>
- [7] The NetBSD Project, “Information about NetBSD 1.5”, <http://www.netbsd.org/Releases/formal-1.5/>

User-level Checkpointing for LinuxThreads Programs*

William R. Dieter and James E. Lumpp, Jr.
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, KY 40506, USA
{dieter,jel}@dcs.uky.edu, <http://www.dcs.uky.edu/~chkpt>

Abstract

Multiple threads running in a single, shared address space is a simple model for writing parallel programs for symmetric multiprocessor (SMP) machines and for overlapping I/O and computation in programs run on either SMP or single processor machines. Often a long running program's user would like the program to save its state periodically in a *checkpoint* from which it can recover in case of a failure. This paper introduces the first system to provide checkpointing support for multithreaded programs that use LinuxThreads, the POSIX based threads library for Linux.

The checkpointing library is simple to use, automatically takes checkpoint, is flexible, and efficient. Virtually all of the overhead of the checkpointing system comes from saving the checkpoint to disk. The checkpointing library added no measurable overhead to tested application programs when they took no checkpoints. Checkpoint file size is approximately the same size as the checkpointed process's address space. On the current implementation WATER-SPATIAL from the SPLASH2 benchmark suite saved a 2.8 MB checkpoint in about 0.18 seconds for local disk or about 21.55 seconds for an NFS mounted disk. The overhead of saving state to disk can be minimized through various techniques including varying the checkpoint interval and excluding regions of the address space from checkpoints.

1 Introduction

Computer systems are prone to hardware and software failures and the probability that a machine will crash be-

fore a process finishes running grows in proportion to the process's run time. A process can save its state in a checkpoint to help tolerate system downtime. A multithreaded process has both private state and shared state. A thread's *private state* includes its program counter, stack pointer, and registers. Its *shared state* includes everything common to all threads in the process, such as the address space and open file state. A multithreaded checkpointing library must save and recover the process's shared state and each thread's private state.

User-level thread libraries are implemented outside the kernel using timers to preempt threads when their time slice is over. Implementing a checkpointing library for a user-level threads package is a straightforward extension of a single-threaded checkpointing library because a user-level multithreaded process is no different from a single-threaded process from the operating system's point of view. User-level threads cannot take advantage of a symmetric multiprocessor (SMP), however, because the kernel is not aware of the threads. Thus it cannot schedule them to run concurrently on separate processors.

With *kernel-level threads*, like LinuxThreads in Linux or lightweight processes in Solaris, the kernel schedules threads and keeps track of their state. Not only must the checkpointing library save and restore the address space of the process to recover the thread state, but it must also call the kernel to restart threads during recovery.

Hybrid thread libraries like the one found in Solaris, use both kernel-level and user-level threads. User-level threads are scheduled to run inside several kernel-level threads, called light-weight processes (LWP) in Solaris. The library usually starts with one LWP per processor. If a user-level thread makes a blocking call and there are more runnable threads the thread library starts a new LWP so the whole process does not need to block. A hybrid thread library cannot be checkpointed like a user-threads library because it uses kernel-level threads.

*This research was supported by the National Science Foundation under Grant CDA-9502645, NFS/EPSCoR under Grant EPS-9874764, the Advanced Research Projects Agency under Grant DAAH04-96-1-0327, and the Kentucky Opportunity Fellowship.

We have tested our checkpointing library on several programs in the SPLASH-2 benchmark suite in addition to some simple test programs. The WATER-SPATIAL application ran with no noticeable overhead other than the time to save a checkpoint. It saved a 2.8 MB checkpoint to local disk in about 0.18 seconds or to an NFS mounted disk in about 21.55 seconds. The time to save a checkpoint to disk is about the same as the time required to copy a file of the same size as the checkpoint with the `cp` command.

Section 2 discusses related work and section 3 describes how programmers and users use the checkpointing library. Section 4 presents the design and implementation of such a library. Section 5 describes restrictions on programs using the checkpointing library. Finally, section 6 shows experimental results and performance.

2 Related work

Checkpointing is a popular way of providing fault-tolerance for computer systems. Both user-level and kernel-level systems have been developed for single threaded processes, however, ours is the first to provide support for multithreaded programs. In addition our system provides this functionality in the form of a user-level library which makes it easier to use and the design is still efficient.

Several other user-level checkpointing libraries for single processes run on multiple versions of Unix [12, 14, 16]. `libckpt` has many features including asynchronous (forked) checkpointing, incremental checkpointing, memory exclusion and user-directed checkpointing [12]. It has been ported to many different versions of Unix. However, `libckpt` does not handle multithreaded processes or dynamically linked executables.

Condor is a process migration system designed to use idle cycles in the network [14]. When the system decides to migrate a process it checkpoints the process on one machine then restarts it on another. Condor runs on a number of operating systems including Solaris and Linux. It neither supports multithreaded programs nor does it have freely available source code.

`libckp` was developed at AT&T Bell Laboratories to checkpoint Unix processes [16]. In contrast with `libckpt`, Condor and our own checkpointing library, `libckp` saves files along with the checkpoint to guar-

antee they will be the same when the program recovers from a checkpoint. Saving copies of all open files guarantees all the files will exist during recovery and allows `libckp` to handle arbitrary file I/O access patterns, but it can make the checkpoint much bigger. Many scientific programs do not need the extra guarantees if the user is willing to retain the input and output files and the application only writes to files in sequential order. `libckp` also does not support multithreaded programs.

Process hijacking uses dynamic executable rewriting to add checkpointing to programs that were not compiled with checkpointing support [19]. Process hijacking does not support multithreaded processes.

MOSIX and `epckpt` provide kernel-level checkpointing solutions. MOSIX is a set of kernel extensions which have been ported to BSD and Linux [4, 3]. MOSIX uses a kernel module to provide transparent load balancing and process migration. `epckpt` is a Linux kernel patch that adds support for processes and process groups [1]. It is in an early stage of development and requires patching, recompiling, and installing a new kernel. Neither MOSIX nor `epckpt` work for multithreaded programs.

Process migration in general [11, 18] is related to checkpointing. Several process migration facilities, like Condor and MOSIX, use checkpointing to provide process migration. In the case of process migration a process is transported through space to another machine. In checkpointing the process is transported to a later time on the same or a different machine. The difference is that a process may recover from a checkpoint at a later time when the environment has changed. Resources the original process was using may be unavailable when it recovers.

Checkpointing for distributed message passing systems has been heavily studied [8]. Most message passing algorithms work to reduce synchronization overhead and handle in transit messages, which are not an issue for multithreaded processes.

The LinuxHA project [2] is bringing support for high availability to Linux. LinuxHA's failure detection mechanisms could be used with our library to automatically restart programs. Most of the LinuxHA work is focused on replicating processes on different machines for fault-tolerance. Replication can offer better guaranteed bounds on recovery time, but usually requires a duplicate machines to take over for each replicated process when a machine fails. Checkpointing only needs extra machines when a machine fails, and then only enough to replace the failed machines. The program can wait until

the the failed machines are repaired if no machines are available and the application can tolerate the delay.

The IEEE Portable Application Services Committee (PASC) 1003.1m Checkpointing Restart working group has been developing a standard API for checkpointing [5].

3 Features

The checkpointing library we introduce here allows, for the first time, LinuxThreads programs to automatically be checkpointed. In addition to checkpointing multi-threaded programs our checkpointing library provides features that help meet our goals of being simple to use, flexible, and efficient.

Adding checkpointing support to a C program is straightforward with our checkpointing library. The application programmer only needs to add one line to include the checkpoint header file:

```
#include "checkpoint.h"
```

and one line to call checkpoint initialization in main.

```
checkpoint_init(&argc, argv, NULL);
```

`checkpoint_init` initializes data structures the checkpointing library uses to track thread and file state. Passing `argc` and `argv` to `checkpoint_init` allows the checkpointing library to read options from the command line. The user can control the checkpoint period by passing optional command line arguments to the checkpointing library. The checkpointing library reads all the arguments after the “--” argument. For example,

```
% prog -- -t period
```

runs the `prog` program with a checkpoint period of `period` seconds. A checkpoint period of 0 disables checkpointing. The user can also pass options to the checkpointing library by putting the options in the `CHKPTOPTS` environment variable. The programmer can set checkpointing options directly using third argument to `checkpoint_init`.

Checkpoints are automatically stored in `prog.chkpt.n` where `prog` is the name of the program and `n` is the

checkpoint number. The user can change the default checkpoint base name with the `-b` option.

To recover from a checkpoint, the user runs the program with the recovery option and specifies a checkpoint file. For example,

```
% prog -- -r prog.chkpt.n
```

runs the `prog` program, loading the state from the checkpointing file `prog.chkpt.n`.

An application program can install callback functions to save any state not saved by the checkpointing library. For example, we have used callback functions to help add checkpointing to the Unify distributed shared memory system [9]. Unify processes communicate through UDP sockets, but the checkpointing library does not save their state. To make checkpointing work Unify makes sure checkpoints are consistent and uses a recovery callback function to reopen the UDP sockets when it recovers from a failure.

A process can install callback functions that will be called before a checkpoint, after a checkpoint, and after recovering from a checkpoint. `chkpt_callback_push` installs three functions: a *pre-checkpoint* callback called before each checkpoint, but after all application threads have been stopped, a *post-checkpoint* callback called after the checkpoint, but before any application thread has been restarted, and a *post-recovery* callback called after recovering from a checkpoint.

Pushing a new set of callback functions does not remove any of the old ones. Instead they are pushed onto a stack. The most recently pushed pre-checkpoint callback function is called last. The most recently pushed post-checkpoint and post-recovery callback functions are called first. The program can remove callback functions in any order using the ID returned by `chkpt_callback_push`. The pushing and popping mechanism simplifies installing and removing callbacks to handle different kinds of state as a program enters different phases.

Our checkpointing library provides memory exclusion similar to that provided by `libckpt` [12]. Memory exclusion allows the application to specify regions of memory that need not be saved in the checkpoint. Excluding large areas of memory that the application does not need reduces the size of the checkpoint.

4 Implementation

The difficulty of checkpointing multithreaded programs comes from making sure that the thread library is in a useful state after recovering from a checkpoint. Threads must be carefully restarted in the correct order to match the way they were originally created.

The basic idea behind our checkpointing library is simple. During initialization the *main thread*, the only thread that exists when the program starts, starts the *checkpoint thread*. After initializing itself, the checkpoint thread blocks with a timed wait on a condition variable. When the timer expires or when another thread calls `checkpoint_now` the checkpoint thread starts a checkpoint. The checkpoint thread is also responsible for running application callback functions.

To take a checkpoint, the checkpointing library blocks all threads, except the main thread, to prevent any threads from changing the process's state while it is being saved. The main thread then saves the process's state and unblocks all the remaining threads.

To recover from a checkpoint, the checkpointing library restarts the threads that were running at the time of the checkpoint. The restarted threads block while the main thread loads the process's state from a checkpoint. Then the main thread unblocks the other threads and they continue running from the checkpoint. Section 4.1 and Section 4.2 describe the algorithm in more detail.

The difficulty comes from doing everything in the correct order, making sure threads do not try to change the address space while it is being saved, and making sure the process's idea of its state matches the operating system's idea of the state. For example the thread library keeps track of the process IDs of all the threads. The checkpointing library must be able to update the thread library's copies of the thread process IDs.

The process state saved in a checkpoint includes the address space, thread registers, thread library state, signal handlers, and open file descriptors. The checkpointing library cannot save every part of the program's state. The unsaved parts lead to the restrictions described in Section 5.

A process's address space is made up of several segments. These segments include the code segment, data segment, heap, stack segment, code and data segments for each of the shared libraries linked with the program, and thread stacks. The checkpointing library uses the

`/proc(4)` file system interface to find the segments that are mapped into memory.

4.1 Saving a Checkpoint

Figure 1 shows how the checkpointing library takes a checkpoint.

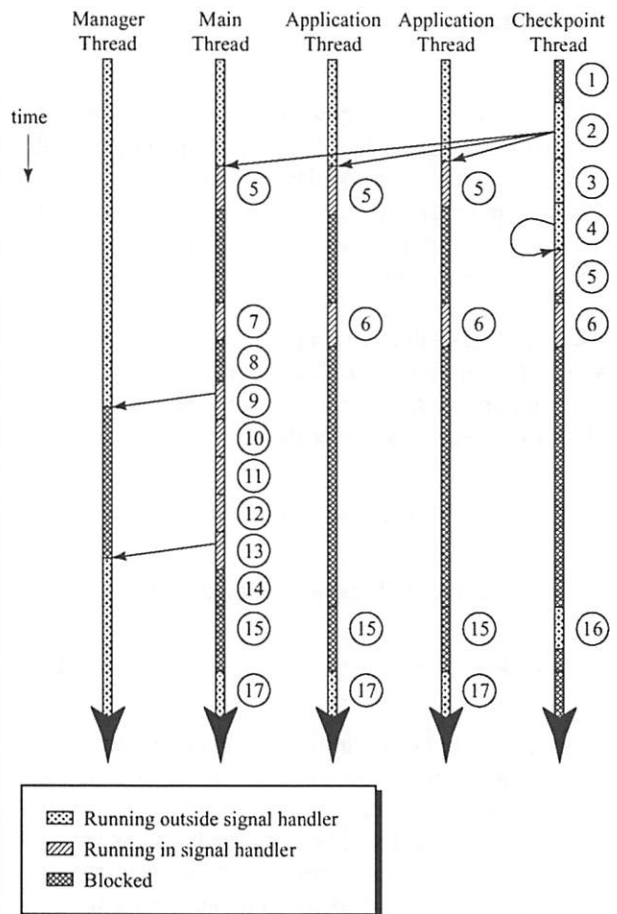


Figure 1: Each thread coordinates with the others to save the process's state. This figure shows how the threads interact.

- 1. Checkpoint thread unblocks.** The checkpoint starts when either the checkpoint thread's timed wait expires or an application thread calls `checkpoint_now`.
- 2. Send a signal to application threads.** To start a checkpoint, the checkpoint thread sends a signal to each of the application threads. Unlike Solaris, when a thread in Linux receives a signal it enters the signal handler for the signal regardless of

the state of the mutex associated with the condition variable [7].

3. **Call pre-checkpoint callbacks.** The checkpoint thread calls the pre-checkpoint callbacks.
4. **Send a signal to the checkpoint thread.** For symmetry the checkpoint thread sends a signal to itself to force itself into its signal handler like all the other threads.
5. **Block signals and wait.** Once in the signal handler every thread blocks all signals and waits at a barrier for the rest of the threads to enter the signal handler.
6. **Save private thread state.** When all threads have entered the signal handler, each thread, except the main thread (and the manager thread), saves its context to memory by calling `sigsetjmp(3)`. Each thread, except the main thread, then blocks at another barrier.
7. **Save signal handlers.** The main thread saves the process's signal handlers using `sigaction(2)`.
8. **Wait for other threads.** The main thread waits until all the other threads have called `sigsetjmp(3)` and reached the barrier.
9. **Stop the manger thread.** The checkpoint thread cannot send a signal to the manager thread when it is signalling all the other threads in step 2 because the manager thread has no thread ID. Instead the main thread sends a message to the pipe the manager thread normally uses to communicate with other threads. When the manager thread receives the message it blocks until the main thread unblocks it.
10. **Save main thread stack environment.** The main thread calls `sigsetjmp(3)` to save its stack environment.
11. **Save file state** Once the other threads have reached the barrier the main thread saves the current file pointer for all open regular files.
12. **Save address space.** The main thread saves the entire address space to the checkpoint file.
13. **Unblock Manager Thread.** The main thread unblocks the manager thread.
14. **Wait at barrier.** The main thread waits at the same barrier as the other threads causing all threads to continue.

15. **Wait at barrier.** After leaving the barrier all threads except the checkpoint thread and manager thread wait at another barrier.

16. **Run post-checkpoint callbacks.** The checkpoint thread runs all registered post-checkpoint callback functions while the rest of the threads wait at the barrier.

17. **Resume execution.** Finally, the checkpoint thread joins the barrier and all the threads leave the barrier, restore their signal mask, and return from the signal handler.

4.2 Restoring From a Saved Checkpoint

When a program recovers from a checkpoint it starts out as a single threaded program. During initialization, the checkpoint library restores the program's state as shown in Figure 2.

1. **Restart threads.** The main thread opens the checkpoint file, reads the saved thread table, and restarts a new thread for each thread in the original program. The thread library automatically creates the manager thread when the checkpointing library creates the first thread.
2. **Restore thread stacks.** The main thread waits while the child threads restore their stack pointers. Each thread restores its stack by calling `siglongjmp(3)` which causes the thread to return from the `sigsetjmp(3)` call it made when it saved its state in the checkpoint. The threads move their stack pointers before the main thread loads the address space because the act of moving them needs to use local variables, which would corrupt the stacks if they were loaded first.
3. **Wait for the main thread.** The child threads wait at a barrier for the main thread to finish restoring the program's state.
4. **Get thread ID to process ID mapping.** After starting all the threads, the main thread calls `pthread_chkpt_restart` to get the new thread ID to process ID mapping. The main thread copies the mapping into an area of memory that will not be overwritten when the main thread restores the address space. `pthread_chkpt_restart` also sends a message to the manager thread telling it to call `siglongjmp(3)` and block so it will be prepared for its stack to be restored.

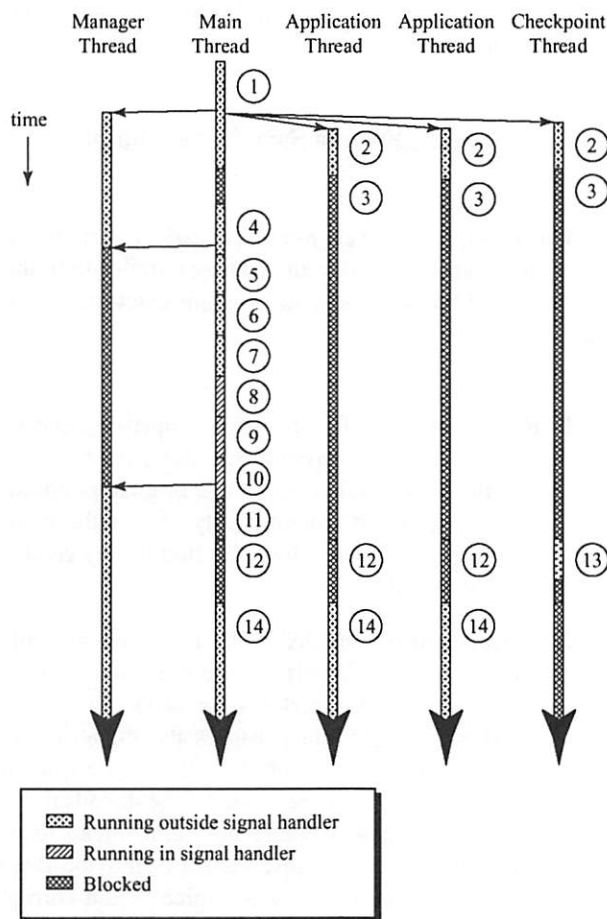


Figure 2: Each thread coordinates with the others to save the process's state. This figure shows how the threads interact.

5. **Load main thread stack.** Once all of the child threads and the manager thread have blocked, the main thread restores its own stack pointer. The main thread stack is not necessarily as large as it was when the the program saved the checkpoint. Therefore the main thread recursively calls a function until the its stack is as large as it was when it saved the checkpoint. The main thread can tell when its stack is large enough by comparing the address of a local variable to the address of a local variable when it saved its state.

6. **Remap the process's address space** The main thread then maps every segment except the main thread stack into the program's address space from the checkpoint file using `mmap(2)` similar to the method Condor uses [10]. The checkpointing library uses `mmap(2)` to remap segments because `mmap(2)` does not cause the data to be loaded immediately. The operating system demand loads the contents of the segments when the program accesses them.

7. **Restore signal handlers.** The main thread restores the signal handlers with `sigaction(2)`.

8. **Restore main thread stack pointer** The main thread calls `siglongjmp(3)` to continue execution where the program was when it saved the checkpoint.

9. **Restore file state** The main thread opens all the files that were open during the checkpoint and moves the file pointer to its position at the time of the checkpoint.

10. **Restore thread ID to process ID mapping.** Next the main thread restores the thread ID to process ID mapping and unblocks the manager thread by calling `pthread_chkpt_postrestart`.

11. **Wait at barrier.** The main thread waits at the same barrier as the other threads causing all threads to continue.

12. **Wait at barrier.** After leaving the barrier all threads except the checkpoint thread wait at another barrier.

13. **Run post-checkpoint callbacks.** The checkpoint thread runs all registered post-checkpoint callback functions while the rest of the threads wait at the barrier.

14. **Resume execution.** Finally, the checkpoint thread joins the barrier and all the threads leave the barrier, restore their signal mask, and return from the signal handler.

4.3 Intercepting Library Functions

The checkpointing library takes advantage of dynamic linking to intercept some library function calls and system calls so it can track the program's state. The checkpointing library intercepts library functions by providing an *intercepting function* with the same name as the library function. The checkpointing library calls `dlsym(3)` with the `RTLD_NEXT` option during initialization to get the addresses of all the intercepted library functions so the intercepting function can call the system version of the function. This method works for system calls as well as library functions because the code to setup and make system calls is part of the C library.

For example, when the application calls `pthread_create(3)`, it gets the checkpointing library's version. The checkpointing library records the parameters passed to `pthread_create(3)` so it can use them during recovery. Then it calls the system `pthread_create(3)` using the address it got from `dlsym(3)` during initialization. If the `pthread_create(3)` call is successful, the checkpointing library updates the number of running threads and returns. Otherwise, it cleans up its thread table and passes the error on to the application program.

4.4 Handling Open File Descriptors

The checkpointing library uses an array that mirrors the kernel's file descriptor table to save the state of open files in each checkpoint. To re-open the files during recovery the checkpointing library needs the filename, mode, and current offset into each open file. When the process opens a file with `open(2)` the checkpointing library adds an entry in its table for that file descriptor with the filename and mode. If the process calls `dup(2)` or `dup2(2)` the checkpointing library links the new file descriptor information to the old file descriptor information. When the process closes the file descriptor its entry is removed from the checkpointing library's file descriptor table. The `read(2)` and `write(2)` system calls are not intercepted.

When the process takes a checkpoint the checkpointing library saves the current file pointer of every regular file. When the process recovers from a checkpoint it uses the information in the checkpointing library's file descriptor table to re-open files and seek the file pointer to the position at the time of the checkpoint.

The checkpointing library intercepts the `popen(2)` call to keep track of pipes that are open. During recovery the checkpointing library reopens pipes to replace those that existed at the time of the checkpoint. However, the checkpointing library does not keep track of data read from or written to the pipe, so data buffered in the kernel may be lost. It also does not handle processes outside the main process. The pipe support is only useful if two threads in the same process share a pipe.

4.5 Linux Specific Issues

Implementing checkpointing for LinuxThreads programs is simpler than for Solaris because LinuxThreads is simpler than the Solaris pthread library. The Solaris kernel treats threads, lightweight processes (LWPs), and processes as different entities. Handling the interactions between threads and LWPs in Solaris is complex. In addition Solaris adds some rules about when a process can handle a signal that complicate the checkpointing library [6].

The Linux kernel is less complex than Solaris because the Linux kernel does not distinguish between threads and processes. LinuxThreads creates threads with `clone(2)` a generalized version of `fork(2)`. Like `fork(2)`, `clone(2)` creates a new process, but `clone(2)` allows the caller to specify which resources the new process shares with its parent and which resources the new process copies from its parent. Thus threads in a LinuxThreads program are separate processes that happen to share an address space and file descriptors with all other threads.

We had to modify the thread LinuxThreads library to handle two different problems. First, the thread library stores a mapping from thread IDs to process IDs in its data segment. When the checkpointing library reloads the process's address space from a checkpoint, the thread ID to process ID mapping is restored to the mapping at the time of the checkpoint, which is out of date for the restored process. To handle this problem, the checkpointing library saves the thread ID to process ID mapping in memory that will not be reloaded from the checkpoint before it restores the address space, but after it restarts threads. The checkpointing library corrects the thread ID to process ID mapping after it restores the process's address space.

Second, the thread library uses a *manager thread* to create processes. When a thread creates a new thread it sends a message to the manager thread through a

pipe and the manager thread creates the new thread. The checkpointing library coordinates with the manager thread during checkpoints to save the manager thread's private state.

The checkpointing library adds four functions to the thread library to handle its interactions with the thread library. The checkpointing library calls `pthread_chkpt_precreate` before it saves the process's address space. `pthread_chkpt_precreate` sends the manager thread a message telling it a checkpoint is beginning. The manager thread saves its environment by calling `sigsetjmp(3)` and blocks. The checkpointing library unblocks the manager thread by calling `pthread_chkpt_postcreate` after saving the checkpoint.

When restoring a process from a checkpoint, the checkpointing library calls `pthread_chkpt_prerestart` to get a copy of the thread ID to process ID mapping and to send a message to the manager thread telling it to call `siglongjmp(3)` and block. The thread library saves the thread ID to process ID mapping in memory that will not be overwritten when the address space is restored. After restoring the address space, the checkpointing library calls `pthread_chkpt_postrestart` to restore the thread ID to process ID mapping and unblocks the manager thread. The `pthread_chkpt_` calls are the only added entry points to the thread library.

5 Restrictions

Our checkpointing library supports programs that access regular files sequentially or use signal handlers for signals. At least one signal must be available for the checkpointing library. The checkpointing library cannot restore process IDs and it does not support programs that randomly access files or communicate with other processes. In most cases, however, the application programmer can add recovery code in callback functions to recover the file or communication state. For example, we are using the checkpointing library to add checkpointing to the Unify distributed shared memory system [9].

Random access reads do not present a problem as long as the program never writes to the file. General random access files are difficult to handle because the checkpointing library must be able to roll the file back during recovery to the state it was in during the checkpoint. One simple way to do this is to save the entire file with the checkpoint [16]. Saving could increase the checkpoint

size a lot if the program uses a lot of large files. The checkpointing library could avoid some of the overhead by not saving files opened with mode `O_RDONLY`. Assuming the files do not change between when they are opened and when the program finishes using them.

The other alternative for handling random access files would be to log each change made to the file. During recovery the checkpointing library could undo all changes made since the last checkpoint. The disadvantage of logging changes is that it adds overhead to log every write operation and the log grows with each write between checkpoints. We did not want to add this overhead when our applications want to use sequential access files. We could reduce overhead by only logging files that are open with mode `O_RDWR`. In our current implementation, an application writer that wants to save random access files with a checkpoint could write a callback routine to manually save the desired files during a checkpoint.

POSIX threads (and LinuxThreads) do not provide a way to create a thread with a particular thread ID. The checkpointing library assumes that the thread library always assigns thread IDs in the same order. As long as that assumption is true, the checkpointing library can guarantee each thread has the same thread ID after recovering by creating threads in the order in which they were originally created. Currently our checkpointing library does not handle programs with threads that exit before the end of the program. If a thread exits early, the thread created immediately after the thread that exited early will get the exited thread's ID during recovery. This problem could be fixed during recovery by creating a thread that exits immediately in place of the exited thread to use up the exited thread's ID. Alternatively we could modify the thread library to allow programs to request particular thread IDs, but we wanted to minimize the changes to the thread library to make it easier to work with different versions of the thread library. The applications we work with create all the threads they need at the start and the threads keep running until the program exits so it was not a problem for our applications.

During recovery, described in section 4.2, the checkpointing library restarts all the threads and restores the process's entire address space from a checkpoint, including the thread library data segment. The checkpointing library assumes that the thread library will function correctly with the newly created threads and the thread data structures from before the checkpoint. This assumption is not entirely true in Linux and thus the thread library must be modified as described in section 4.5.

The checkpointing library interrupts each thread with a

signal to start a checkpoint. When the checkpointing library installs its signal handler, it passes `SA_RESTART` flag to `sigaction(2)` to tell the Linux kernel to restart interrupted system calls if possible. The application code must restart system calls that the Linux kernel cannot restart.

We added nothing extra to support thread cancellation functions. The problem with thread cancellation is recreating the threads with the correct thread IDs as described above. Otherwise the checkpointing library would just need minor adjustments to cleanup the canceled thread after its last cancellation handler is called.

We also did nothing special to support thread scheduling priorities. Handling thread scheduling priorities would be a matter of logging the calls to the thread scheduling priority calls and reissuing them to restore scheduling priorities during recovery. Support may be added if there is demand.

The checkpointing library does not handle interprocess communication, but it does reopen pipes open at the time of a checkpoint. It cannot make another process use either end of the pipe it opens, and it does not save any data buffered in the kernel. Thus pipes will only be restored if they are used between threads in the same process and no data is buffered in the pipe at the time of the checkpoint.

Handling IPC, either between processes on the same machine or on different machines, is difficult in general. Both processes must agree on when they take checkpoints or make assumptions about how deterministic they are to avoid inconsistent checkpoints. Otherwise one process could fail and recover from a checkpoint that rolls it back to a state in which it has not sent a message on which another process depends. At that point the program is in a state which it could not have reached without a failure. The second process is in a state the causally depends on a state that never happened, as far as the first process is concerned. In that case the two processes are said to be *inconsistent*. Issues of consistency have been well studied and are beyond the scope of this paper [8]. Extending our checkpointing library to work for a general case with multiple processes communicating with pipes or TCP sockets would be non-trivial.

The checkpointing library does not intercept or modify time related system calls in any way. A program that uses absolute time values may behave strangely after recovering from a checkpoint. For example, assume a thread blocks using `pthread_cond_timedwait(3)` to wait for several seconds and the process checkpoints

while the thread is blocked. The process is killed and restarts from the checkpoint several minutes later. After recovery, the thread will immediately unblock because the timer has expired.

Strictly speaking this behavior is correct because the current time is later than the time for which the thread was waiting. Applications that have time based events, however, might get a flood of expired timers when recovering from a checkpoint. We could intercept all calls that have anything to do with absolute time and adjust the time they see after recovering from a checkpoint, but some programs need to know what the absolute time really is. Instead we leave it up to the application programmer to write a callback function to adjust any time values that need to be adjusted after recovering from a checkpoint.

6 Results

The checkpointing library adds overhead due to intercepting calls, overhead due to the checkpointing thread, and overhead due to saving checkpoints. The checkpointing library only intercepts thread creation, file open, and file close calls. Unless the program opens and closes files often or creates and destroys threads often that overhead will be low. The checkpointing thread does not add much overhead because it is blocked except during checkpointing.

6.1 Applications

To verify that checkpointing adds little overhead when it is not writing a checkpoint, we ran several applications from the SPLASH2 [17] benchmark suite. SPLASH2 is a set of benchmarks designed to test the performance parallel shared memory machines. The benchmarks are based on applications and kernels commonly used in scientific computing. We present the results of running the BARNES and WATER-SPATIAL benchmarks below. The other SPLASH2 benchmarks we ran gave similar results.

BARNES simulates the gravitational effects of a number of bodies in space using the Barnes-Hut algorithm. It uses a tree to represent the locations of the bodies in space. WATER-SPATIAL simulates the interaction of water particles using a 3 dimensional grid. We increased the problem sizes of both applications from the size used in the original SPLASH2 paper [17] to in-

crease the running time of the benchmarks. For WATER-SPATIAL we increased the number of particles to 8000 instead of 512. For BARNES we used 65536 particles instead of 16384.

The test machine was a two processor 500 MHz Pentium III SMP machine with 512 KB L2 cache and 128 MB of main memory running Linux kernel version 2.2.10 with NFS v2 and glibc2. The file systems mounted for the NFS tests were served by an UltraSPARC-10 running Solaris 5.7. The network over which the NFS disk was mounted was a moderately used 100 Mb/s switched Ethernet connected by a Cisco Catalyst 2924 XL auto-sensing 10/100 Mb/s switch. None of the machines used in the test shared a port on the switch with any other machine.

Table 1 compares running each of the benchmarks with and without checkpointing linked for one and two processors. With checkpointing linked, the program called the checkpoint initialization code, but did not take any checkpoints. This test shows how much overhead checkpointing adds not including the time to save the checkpoint to disk. Table 1 shows that the checkpointing library does not add much overhead when the program is not checkpointing. This overhead is important because the program will spend most of its time running, not checkpointing.

In some cases, code with checkpointing linked but not used runs faster than without checkpointing. Changes in the program's memory layout cause this speedup. Linking the benchmark with unused code gave the same effect as linking with the checkpointing library.

Table 2 shows the amount of time BARNES and WATER-SPATIAL spend taking a checkpoint using a local disk or an NFS mounted disk. Most of the time is spent saving the checkpoint to a file. These numbers are intended to give a feel for how long it takes for application programs to checkpoint and how the checkpointing time is spent. In both the local disk and the NFS disk cases the amount of time spent synchronizing threads before and after the checkpoint is several orders of magnitude smaller than the amount of time to save the checkpoint to disk. Saving the checkpoint to disk is the largest overhead of checkpointing.

6.2 Checkpoint Size

The size of the checkpoint file is directly proportional to the size of the process's address space. Most of the

overhead of taking a checkpoint comes from writing the address space to disk. Figure 3 gives an idea how long a program will take to save a checkpoint depending the size of the checkpoint and whether it is saving to a local disk or an NFS mounted disk.

The amount of time required to save the checkpoint depends on the file system to which it is saved. The figure shows times for local disk and NFS. Writing a checkpoint to an NFS mounted disk¹ takes much longer than writing to a local disk. The figure also shows that the amount of time required to save a checkpoint is directly proportional to the size of the checkpoint.

In this case, the time required to save a checkpoint to the NFS mounted disk can be approximated by the equation $t_{cp} = 2.37\text{sec/MB} \times s - 0.60\text{sec}$ for checkpoints larger than about 1.3 MB, where s is the size of checkpoint. The user can use t_{cp} to decide which checkpoint interval to use. A simple method is to calculate the maximum percentage of execution time taken by checkpointing given t_{cp} and the maximum address space size (checkpoint size) of the program. More sophisticated methods can determine the checkpoint interval that will minimize the program's expected run time given t_{cp} and a particular failure rate [13, 15].

7 Conclusion

Our checkpointing library provides checkpointing multithreaded Linux programs. It adds little overhead except when taking a checkpoint. The overhead that it does add is directly proportional to the size of the address space. Saving the checkpoint to local disk is much faster than saving it to an NFS mounted disk.

Our checkpointing library combines simplicity for many programs but flexibility for programs willing to use it. Callback function provide flexibility for applications that have special needs. Features like memory exclusion can and user directed checkpointing can reduce overhead when taking a checkpoint. We are considering adding incremental and asynchronous checkpointing to further reduce the overhead of saving a checkpoint.

The latest version of the checkpointing library is available through our web site at:

<http://www.dcs.uky.edu/~chkpt>

¹The disk with mounted with $rsize=8192, wsize=8192$.

<i>Benchmark</i>	<i>1 Processor</i>			<i>2 Processors</i>		
	Not Linked (seconds)	Linked (seconds)	Overhead (percent)	Not Linked (seconds)	Linked (seconds)	Overhead (percent)
BARNES	53	53	0	32	31	-3.1
WATER-SPATIAL	666	678	1.8	335	340	1.5

Table 1: This table gives execution times of several SPLASH2 applications. For the “Not Linked” case, the application was run without the checkpointing library linked to it. In the “Linked” case, the checkpointing library was linked to the application, but it did not take any checkpoints. This case measures checkpointing overhead outside not directly related to taking checkpoints. All times are in seconds.

<i>Benchmark</i>	<i>Local Disk</i>			<i>NFS</i>		
	Total Time (seconds)	Save Time (seconds)	Sync Time (seconds)	Total Time (seconds)	Save Time (seconds)	Sync Time (seconds)
BARNES	9.8351	9.8330	0.0021	337.1552	337.1536	0.0017
WATER-SPATIAL	0.0736	0.0716	0.0020	16.3965	16.2361	0.1604

Table 2: This table lists the time required to save checkpoints for several SPLASH2 applications. “Total Time” is the total amount of time to take a checkpoint. “Save Time” is the amount of time required to save the checkpoint to a file. “Sync Time” is the amount of time required for processes to synchronize before and after the checkpoint. All times are in seconds with 2 Processors.

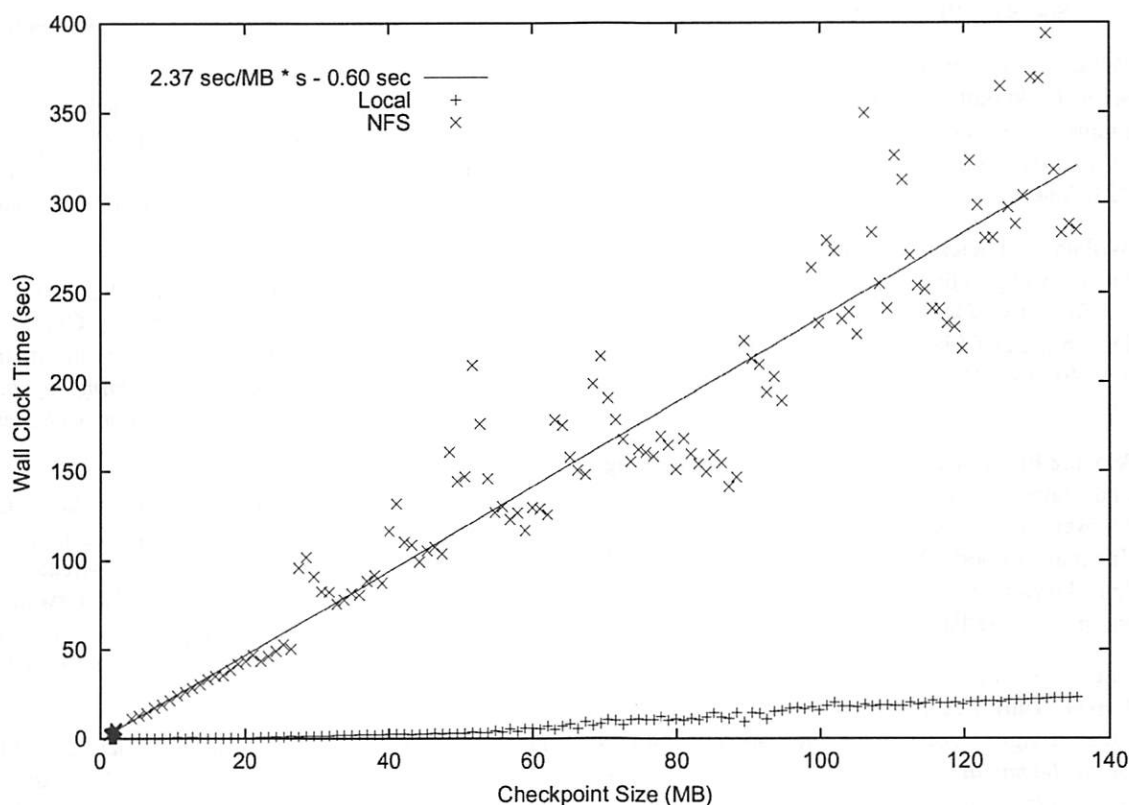


Figure 3: This graph shows a plot of the time to save a checkpoint as a function of checkpoint size. The line through the NFS data points was fitted using the method of least squares.

References

- [1] EPCKPT: Eduardo Pinheiro Checkpoint Project Website. Technical report, <http://www.cs.rochester.edu/u/edpin/epckpt/>.
- [2] High-Availability Linux Project Website. Technical report, <http://www.linux-ha.org/>.
- [3] Amnon Barak, Oren La-adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for LINUX. In *Proceedings of Linux Expo '99*, pages 95–100, May 1999.
- [4] Amnon Barak and Aren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998.
- [5] Portable Application Standards Committee. *Draft Standard for Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) – Amendment m: Checkpoint/Restart Interface*. IEEE Computer Society, p1003.1m/d2 edition.
- [6] William R. Dieter and James E. Lumpp, Jr. A user-level checkpointing library for POSIX threads programs. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 224–227, June 1999.
- [7] William R. Dieter and James E. Lumpp, Jr. User-level checkpointing of posix threads. Technical Report CEG-99-004, University of Kentucky, Department of Electrical Engineering, Lexington, KY 40506–0046, <http://www.dcs.uky.edu/~chkpt>, 1999.
- [8] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and Davaid B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, <http://www.cs.utexas.edu/users/lorenzo/papers/Pap6.ps>, June 1999.
- [9] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Unify: A scalable approach to multicomputer design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(2), 1995.
- [10] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Science, April 1997.
- [11] Dejan S. Milojičić, Fred Douglass, Yves Paindavaine, Richard Wheeler, and Songnian Zhou. Process migration survey. *ACM Computing Surveys*, pages 241–299, September 2000.
- [12] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *USENIX Winter 1995 Technical Conference*, January 1995.
- [13] James S. Plank and Wael R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 48–57, June 1998.
- [14] Todd Tannenbaum and Michael Litzkow. Checkpointing and migration of unix processes in the condor distributed processing system. *Dr. Dobbs Journal*, pages 40–48, 102, February 1995.
- [15] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, pages 942–947, August 1997.
- [16] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and its applications. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 22–31, June 1995.
- [17] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, June 1995.
- [18] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabbii, and Durriya Netterwala. An OSF/1 Unix for massively parallel multicomputers. In *USENIX Winter 1993 Technical Conference*, pages 449–468, January 1993.
- [19] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, pages 177–184, August 1999.

Building an open-source Solaris-compatible threads library

John Wood
Compaq Computer (UK) Ltd
woodjohn@compaq.com

Abstract

Applications that use the Solaris threads application programming interface (API), e.g. *thr_create()*, *mutex_lock()*, *cond_signal()*, etc. [1], are generally non-portable. Thus to port an application that uses Solaris threads to another platform will require some degree of work.

Solaris now supports the POSIX threads API as well as the Solaris threads API. Therefore to make a Solaris threaded application portable, the ideal is to re-code the threaded part of the application to use POSIX threads. However, the Solaris threads API has some unique functionality over the POSIX threads API. This can make the task of converting a Solaris threaded application to use POSIX threads be very time-consuming and hence expensive, sometimes prohibitively so.

This paper outlines an alternative approach to porting applications that use the Solaris threads API, which is to use an open-source Solaris-compatible threads library that layers upon a POSIX threads library. The objective is to allow an otherwise-portable Solaris threaded application to be ported by simply rebuilding on the target platform using the Solaris-compatible threads library and header-files. This reduces the cost of porting the application.

1 Introduction

1.1 Porting applications from Solaris

Many independent software vendors (ISVs) develop applications on Sun's Solaris platform. These applications may have used the Solaris threads¹ API: this is most likely for mature applications that pre-date the POSIX threads² API.

To port an application that uses the Solaris threads API to another UNIX platform typically requires re-working the application to use the POSIX threads API. Depend-

ing upon the application, the amount of re-working needed may vary from simply being a matter of a few editor substitutions, to re-engineering the application.

An alternative to requiring every Solaris threads application to be re-worked for portability is to provide a portable Solaris-compatible threads library. Once the Solaris-compatible threads library has been ported to a target platform, threaded Solaris applications may be ported by just re-compiling.³ This paper describes a Solaris-compatible threads library, for which the acronym STL is used. An ISV porting a threaded Solaris application may choose not to use STL, but may find the details of the STL implementation useful to re-work their application for portability.

1.2 Objective

The objective of STL is to provide a high degree of Solaris threads compatibility on the target platform for the minimum amount of effort. To reduce effort and maximize portability, STL is layered upon the POSIX threads library. This limits what STL is capable of, but this was considered an acceptable compromise rather

¹ The Solaris threads API is a subset of the UNIX International (UI) threads API, which is also supported by SCO's UnixWare 2.

² *POSIX threads* refers to the thread-specific part of the formal standard ISO/IEC 9945-1:1996 [POS96] that is commonly known as POSIX 1003.1-1996. This standard integrates the original POSIX 1003.1-1990 [POS90] standard (base operating system API) with the amendments 1003.1b-1993 (real-time extensions) and 1003.1c-1995 (threads). See *Threaded Programming Standards* [2] and the *comp.programming.threads* FAQ [3] for further information.

³ But see also the section *Non-objectives* for a description of other potential porting issues.

than trying to implement a fully compatible Solaris threads library from scratch.

1.3 Non-objectives

The scope of STL is limited to providing the Solaris threads API. STL does not attempt to solve generic porting issues such as:

- Differences in system and library calls.
- Architectural differences. E.g. 32 to 64 bits; endianism, etc.
- Compiler and other build-tool differences.

Platform-specific porting guides such as the *Sun Solaris to Compaq Tru64 UNIX Porting Guide* [4] cover these sorts of issues.

1.4 Solaris Compatibility Libraries for Tru64 UNIX

STL was originally developed as part of the *Solaris Compatibility Libraries (SCL)* [5] for Compaq's Tru64 UNIX operating system. SCL was developed to ease the porting of Solaris applications to Tru64 UNIX.

SCL v1.1 was released in April 2000, and is available for free download. SCL v1.1 provides:

- Solaris Threads: An implementation of Solaris thread functions layered upon POSIX threads.
- Remote Procedure Calls: A port of Sun's freely redistributable ONC RPC v2.3 software.
- Miscellaneous: Various library functions including asynchronous I/O, large file support, wide character and signal name functions.

SCL is supplied as-is, with no formal support. The SCL source-code is freely available, and is included in the download kit.

2 Implementation strategy

STL is written in the C programming language, and is compiled to produce a shared object library, which is supplied with two header files.

2.1 Comparing the Solaris threads API to the POSIX threads API

Most threads APIs provide similar functionality of being able to create and manage a thread, and to create and manage synchronization objects. The Solaris threads API supports mutexes, condition variables, read/write locks and semaphores as thread synchronization objects. POSIX threads supports most of these synchronization objects⁴, but does not include read/write locks, which are an extension from the Single UNIX Specification Version 2 (SUSv2) [UX98] for the UNIX 98 brand⁵. Therefore many UNIX systems that support POSIX threads also support read/write locks.

In many cases the Solaris threads API and POSIX threads API are almost identical. For example, consider the *kill* function that sends a signal to another thread:

Solaris threads API:

```
int thr_kill(
    thread_t  target_thread_id,
    int       sig );
```

POSIX threads API:

```
int pthread_kill(
    pthread_t  target_thread_id,
    int       sig );
```

Both functions take the same parameters of a thread identifier and signal number, although the thread identifier type is different. Both functions return an integer value, which if 0 indicates success.

In essence, to implement *thr_kill()*, STL first defines the Solaris type *thread_t* to match the POSIX threads type *pthread_t*, and then *thr_kill()* just becomes a jacket routine to *pthread_kill()*.

Mapping the Solaris threads types directly onto their POSIX threads equivalents potentially allows an application to mix Solaris thread API calls with POSIX thread API calls. However, STL does not currently support this. To support mixing would require STL to intercept application calls to some of the POSIX threads functions for thread management, such as:

⁴ Actually, semaphores were defined by POSIX 1003.1b-1993 (real-time extensions) rather than POSIX 1003.1c-1995 (threads).

⁵ Specifically from the *X/Open CAE Specification, System Interfaces and Headers, Issue 5* (also known as XSH5) part of SUSv2.

- `pthread_create()`
- `pthread_join()`
- `pthread_detach()`

so that STL's internal data structures are properly maintained. But mixing calls from the different APIs for synchronization objects should not be an issue.

For most Solaris threads functions, the STL jacket routines have to perform extra work to ensure compatibility. Two examples are given later in this section.

There are a number of areas where Solaris threads provides functionality that is not available within POSIX threads. The main areas are:

- Daemon threads.
- Joining any thread.
- Suspending and continuing a thread.

Each of these areas is described in a separate section. There is also a section on getting and setting information of another thread, which became necessary to implement several pieces of Solaris threads functionality.

For details of STL's functionality and restrictions, see the *SCL Users Guide* [6]. Specifically, see section 3.2: *STL Functionality*; Appendix A: *Mapping of Solaris thread types to POSIX thread types by STL*, and Appendix B: *Solaris thread functions implemented by STL*.

2.2 STL design issues

The following decisions were made when designing and implementing STL:

- Always try to be compatible with Solaris threads, even if this may impact performance.
- Be compatible with both the documented behaviour and the undocumented, observed behaviour of Solaris threads.

Two examples are now described.

2.2.1 Function return values

STL implements many Solaris thread functions by calling the equivalent POSIX threads routine. But often the function return values documented for the two APIs differ. STL handles this using the following rules:

- If the STL implementation of a Solaris routine receives an error value from calling a POSIX threads function, and that value matches one of the error values documented for the Solaris routine, then it just returns that value.
- If the STL implementation of a Solaris routine receives an error value from calling a POSIX threads function that does not match one of the error values documented for the Solaris routine, then it maps the value to a valid Solaris error value for that function, logs a message⁶, and returns the mapped value.

Thus a Solaris application does not need changing to handle the potentially different error value returns from POSIX threads functions, because STL handles this. But be aware that the reasons for a particular error value may be quite different when using STL.

For example, consider the Solaris threads and Tru64 UNIX POSIX threads functions to perform a timed-wait on a condition variable. The APIs are:

Solaris threads API:

```
int cond_timedwait(
    cond_t      *cvp,
    mutex_t     *mp,
    timestruc_t *abstime );
```

Function return values documented on Solaris:

0: successful completion.

EFAULT: a parameter has an invalid address.

EINVAL: invalid *abstime* parameter: if *abstime* is more than 100,000,000 seconds in the future, or the nanoseconds field of *abstime* is $\geq 1,000,000,000$.

ETIME: the time specified by *abstime* has passed.

The POSIX threads API:

```
int pthread_cond_timedwait(
    pthread_cond_t      *cond,
    pthread_mutex_t     *mutex,
    const struct timespec *abstime );
```

⁶ See the *Error Logging* chapter of the *SCL Users Guide* [6]. Requires setting the *SCL_LOG_FILE* environment variable.

Function return values documented on Tru64 UNIX:

0: successful completion.

EINVAL: the value specified by *cond*, *mutex* or *abstime* is invalid, or: the mutex was not owned by the calling thread at the time of the call, or: different mutexes are supplied for concurrent *pthread_cond_timedwait()* or *pthread_cond_wait()* operations on the same condition variable.

ETIMEDOUT: the time specified by *abstime* has passed.

ENOMEM: the POSIX threads library cannot acquire the memory needed to block using statically initialised objects.

The Solaris *cond_timedwait()* function can return a value of 0, EFAULT, EINVAL or ETIME. The *pthread_cond_timedwait()* function on Tru64 UNIX can return 0, EINVAL, ETIMEDOUT or ENOMEM.

If the STL implementation of *cond_timedwait()* receives an ENOMEM return value from calling *pthread_cond_timedwait()*, then it maps this value to EFAULT, which is one of those documented for the Solaris function. This is because Solaris applications might only check for specific error codes, rather than just testing the status for success. Additionally, STL logs a message to indicate when it is performing a mapping of error statuses (e.g. “ENOMEM from *pthread_cond_wait()* mapped to EFAULT from *cond_timed_wait()*”): these messages may be helpful to understanding the real reason for a particular STL function return value.

Note that a message is only logged when the POSIX threads function’s return value is mapped to a different return value for the Solaris threads routine. For example, with *cond_timedwait()*:

- If STL’s *cond_timedwait()* is called with an uninitialized mutex, then EINVAL is returned from *pthread_cond_timedwait()* which is passed back from *cond_timedwait()* with no message logged because there is no mapping of errors (even though the reason for the EINVAL is different to what Solaris documents).
- If STL’s *cond_timedwait()* receives ETIMEDOUT from *pthread_cond_timedwait()*, then it maps this to ETIME and logs a message indicating the mapping (even though the error codes have the same meaning).

2.2.2 Documented and observed behaviour of synchronization objects

Solaris documents that synchronization objects that are statically initialized to all zeros do not need to be explicitly initialized. It does not define the result when attempting to use uninitialized objects as function parameters: the observed behaviour is that Solaris implicitly initialises the object. This is consistent with the Solaris routines not having the error return EINVAL defined for uninitialized objects.

POSIX threads documents and returns EINVAL when uninitialized objects are used as parameters.

There is no portable way to validate that a POSIX threads synchronization object has been initialized, so STL does not support Solaris’ observed behaviour of implicitly initializing uninitialized synchronization objects. Attempts to use an uninitialized object with STL functions results in an error value being returned, and typically an error-mapping message is logged. But there is an exception for statically-initialized-to-zero objects: STL tests for these, and will explicitly initialize them, to conform to Solaris’ documented behaviour. These checks will impact performance, but compatibility is the main objective.

3 Daemon Threads

3.1 Introduction

Solaris threads provides the concept of *daemon* threads. Solaris threads defines that a process terminates when its last non-daemon thread terminates.⁷

POSIX threads does not support daemon threads. A POSIX threads process terminates when its last thread exits.

Daemon threads could be used by applications for housekeeping tasks. For example, a daemon thread may be created that periodically monitors disk space whilst the application is running.

⁷ A threaded process will also terminate if any thread calls *exit()*, either explicitly, or, for the main thread only, implicitly if the main thread finishes without calling *pthread_exit()* (POSIX threads) or *thr_exit()* (Solaris threads).

3.2 Implementation

Conceptually, to implement support for daemon threads is quite simple: a count of the number of non-daemon threads must be maintained. This count may need adjusting when a thread is created, or when a thread terminates. If the non-daemon thread count becomes zero, then the process must be terminated.

The daemon-thread implementation sounds simple in outline, but now we look at the implementation in more detail. It requires:

- Keeping a global count of the number of non-daemon threads.

Updates to the count need to be coordinated by using a mutex.

- Knowing when a thread is created, and whether it is a daemon or not.

A daemon thread is created when the *THR_DAEMON* bit is set in the flags parameter to *thr_create()*.

- Knowing when a thread terminates, and whether it is a daemon or not.

Adjust non-daemon thread count if appropriate; if count is now zero, terminate the process.

The tricky bit here is the last bit: to know when a thread terminates, and to determine if it is a daemon-thread or not. The non-daemon thread count has to be decremented if the terminating thread is not a daemon. Thread-specific data, along with a destructor-routine, is used to implement this. This works as follows.

When STL initializes, it creates a thread-specific data key *STL.tsd_key*, and associates a destructor routine, *stl_tsd_key_destructor()*, with that key. When a thread that has data associated with *STL.tsd_key* terminates, it runs the *stl_tsd_key_destructor()* routine.⁸

When a new thread is created by calling *thr_create()*, the STL implementation of *thr_create()* dynamically allocates some memory *M* for a *stl_tsd_t* data structure, and fills in its fields. The *stl_tsd_t* structure has fields for:

⁸ An important feature of the thread-specific data destructor routine is that this routine always gets called, regardless of how the thread terminates.

- The flags parameter to *thr_create()*.
- The *start-routine* parameter to *thr_create()*.
- The *arg* parameter to *thr_create()*.

STL's *thr_create()* then calls *pthread_create()* but specifies *stl_thread_start_rtn()* as the start-routine parameter, and *M* as the start-routine argument.

When the new thread starts, it first executes *stl_thread_start_rtn(M)*. Within this routine the thread makes the memory *M* into thread-specific data for this thread by calling *pthread_setspecific(STL.tsd_key, M)*. The new thread also determines from the thread's attribute flags in *M* if it is a daemon thread, and if not then the count of non-daemon threads is incremented.

The new thread then calls the user-specified start-routine with the user-specified argument, which are both extracted from *M*.

When the new thread terminates, it automatically executes the *stl_tsd_key_destructor()* routine. Ultimately, this frees up the memory *M*, but first it extracts the thread's attribute-flags from *M*, and determines from the flags whether this thread is a daemon thread or not. If the thread is not a daemon, then the count of non-daemon threads is decremented; and if this count is now zero, then process is terminated by calling *exit()*.

3.3 Daemon thread considerations

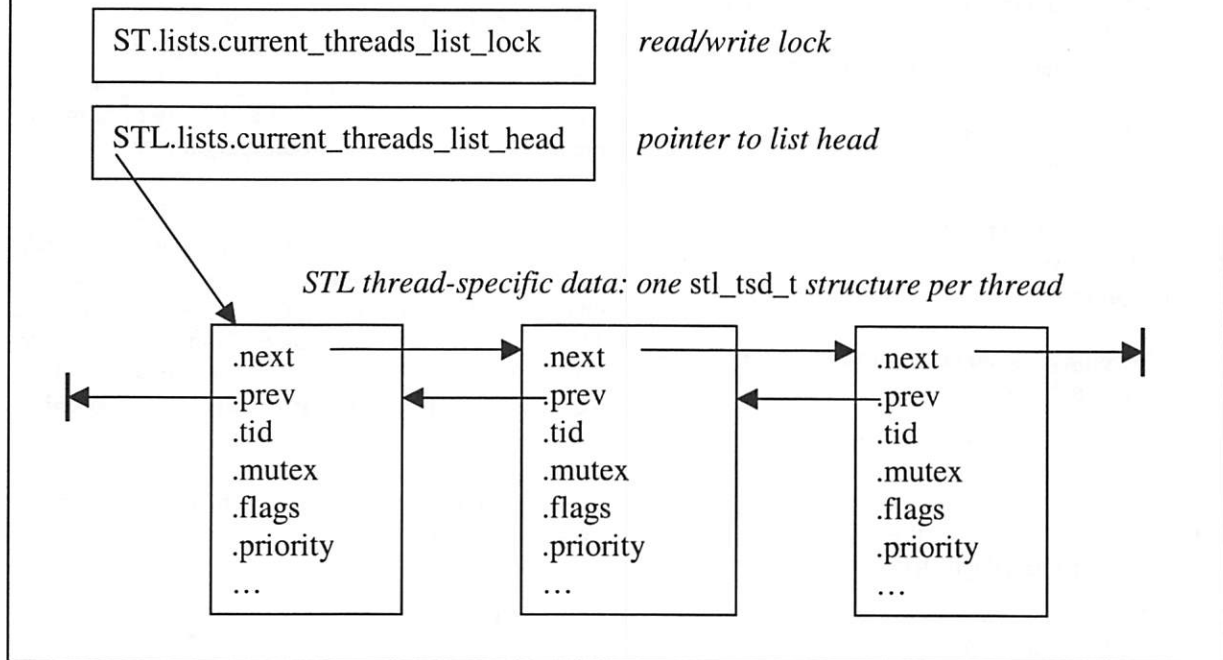
On Solaris, when a Solaris-threaded process forks, the child process has a copy of all the threads, whereas with POSIX threads the child only has one thread. Hence the STL count of non-daemon threads needs resetting. This is implemented by calling *pthread_atfork()* to declare a routine which is invoked by the child process after a fork, that resets the thread count.

For STL to support the mixing of Solaris threads and POSIX threads calls, which Solaris allows, it would need to intercept calls to *pthread_create()* to increment the non-daemon thread count.

4 Getting and setting information of another thread

Normally a thread only needs to access its own thread-specific data, which it can readily find by calling *pthread_getspecific()*. However, there are a few in-

Figure 1. List of current threads.



stances when it is necessary for one thread to access another thread's STL thread-specific data. For example:

- when `thr_getprio()` or `thr_setprio()` are called, to get/set another thread's priority.⁹
- when `pthread_detach()` is called, to make the target thread detached.

So a mechanism is needed whereby one thread can find and access another thread's STL thread-specific data.

4.1 List of Current Threads

STL maintains a list of all the current threads in the process. One thread can search the list of current threads to find and subsequently access another thread's STL thread-specific data.

The list of current threads is currently implemented as an unsorted linked list. The STL thread-specific data-structure `stl_tsd_t` is extended to make it be an element of the linked-list by adding pointers to the next and pre-

vious entries in the list. These two pointers make it quicker for a thread to unlink itself from the list. The `stl_tsd_t` structure is also extended to include the thread identifier `tid` of that thread, and a mutex for that thread. If one thread wants to get or set another thread's attributes, then it must lock the target thread's mutex.

A global pointer, `STL.lists.current_threads_list_head`, points to the head of the list of current threads. New thread-list entries are added to the head of the list. The list will never be empty, except when the last thread of a process is terminating.

Access to the list is coordinated by a read/write lock, `STL.lists.current_threads_list_lock`. The read/write lock gives better concurrency than a mutex, because write-access to the list is only required when a thread is being created (added to the list) or terminating (removed from the list).

Figure 1 illustrates the current-threads list.

Only the thread itself can add itself to the list of current threads, or remove itself from the list of current threads. This has implications for the locking assumptions. A new thread adds itself to the current-threads list by executing code within the `stl_thread_start_rtn()` routine. A thread removes itself from the list when it terminates by executing code within the `stl_tsd_key_destructor()` routine.

⁹ Solaris provides functions to set and get a thread's priority. Since the POSIX threads API has no direct equivalent to explicitly get or set a thread's priority, STL only stores a thread's priority so that `thr_getprio()` returns the value set by `thr_setprio()`.

The STL implementation currently uses a linear search to locate a specified thread's *stl_tsd_t* entry. This may have poor-performance implications when the number of current-threads is large, but this is considered acceptable based on the premise that it is not that often that one thread modifies another thread's *stl_tsd_t* entry. Usually a thread will modify its own *stl_tsd_t* entry, which it finds quickly by calling *pthread_getspecific()*. Thus to implement a function like *thr_setprio(tid,pri)* the sequence of events is:

- Lock the current-threads-list for reading.
- Search the current-threads-list, starting from the list-head:
 - If list-entry's thread-ID matches the one we're looking for, then:
 - Lock that thread's mutex
 - Access that thread's data
 - Unlock target thread's mutex
 - Set *return_status* to indicate success
 - Break out of search
 - Else:
 - Move on to the next thread's entry in list
 - If this is the end of the list
 - Set *return_status* to indicate thread not found (ESRCH)
- Unlock the current-threads-list
- Return *return_status*

5 Joining any thread

5.1 Introduction

When a non-detached Solaris or POSIX thread terminates, it should be joined to obtain its return-status, and to enable the threads library to release any resources that were not released when the thread terminated. Failure to join a non-detached thread results in a *zombie*¹⁰-like thread, which can cause memory leaks.

Compare the Solaris threads and POSIX threads APIs for joining a thread:

Solaris threads API:

```
int thr_join(
    thread_t tid,
    thread_t *ret_tid,
    void **ret_val );
```

POSIX threads API:

```
int pthread_join(
    pthread_t tid,
    void **ret_val );
```

Both the Solaris threads and POSIX threads APIs let you join with a specific thread identified by *tid*. In addition, if you specify a thread identifier *tid* of 0 on Solaris, *thr_join()* will join with any terminated non-detached thread that has not yet been joined, and will return the identifier of the joined thread in *ret_tid*. If there are no non-detached terminated threads waiting to be joined, then *thr_join()* with a *tid* of 0 will wait for the first such thread to terminate, and will join with it. This unique Solaris functionality is called *join-any-thread*.

Note that when a joinable thread terminates, there may be zero, one or many threads waiting to join that specific thread, as well as other threads waiting to join any thread. Solaris does not define the behaviour for this situation, but STL gives preference to the thread(s) waiting to join the specific thread over the threads waiting to join an unspecified thread.

5.2 Implementation

Three new lists are used by STL to implement join-any-thread:

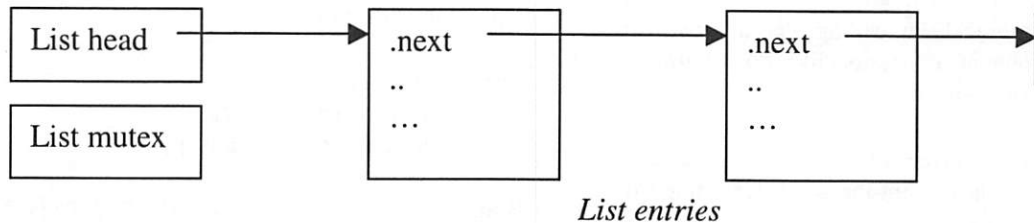
1. List of threads waiting to join a specific thread (*join-specific list*).
2. List of threads waiting to join any thread (*join-any list*).
3. List of joinable threads that have terminated and are awaiting joining (*terminated list*).

The format of these lists is illustrated in Figure 2. Code is added to *thr_join()* and to the STL thread-specific data destructor routine to maintain these lists. The pseudo-code is shown below.

¹⁰ In UNIX, a zombie process is a process that has terminated but has not been reaped by the parent process calling one of the *wait()* system calls.

Figure 2. Lists to implement *join-any-tread*.

Three lists, all follow the same format (immediately below), but contain different data (see table below).



Contents of list entry:

<i>Join-specific list</i>	<i>Join-any list</i>	<i>Terminated list</i>
<ul style="list-style-type: none"> ID of specific thread being waited upon to join with 	<ul style="list-style-type: none"> ID of thread to join with,. Initially set to 0. Used by waiter as a predicate for a <i>pthread_cond_wait()</i> loop Mutex Condition Variable 	<ul style="list-style-type: none"> ID of terminated thread

5.2.1 To join with a specific thread:

- If (specific thread exists in list of current threads) and (specific thread is joinable [not detached])
 - Add an entry for this thread to the join-specific list
- pthread_join(specific_thread, ret_val)*

5.2.2 To join with any thread:

- If (terminated-list is not empty)
 - Remove entry from head of terminated-list; extract terminated thread ID
- Else
 - Add a new entry with *tid=0* to the join-any list
 - Block until the join-any list entry contains a suitable terminated thread ID to join with
 - Remove our entry from the join-any list
- pthread_join(terminated thread ID, ret_val)*

5.2.3 When a thread terminates:

- If (terminating thread is detached)
 - Return */* thread is not joinable */*
- If (join-specific list has any entries that match this thread's ID)
 - Remove all matching entries from the list
- Else if (join-any list is not empty)
 - Put terminating thread's ID into the entry at head of list, and unblock the waiting thread
- Else
 - Add this thread's ID to the terminated list
- /* thread now finally terminates */*

6 Suspending and Continuing a thread

6.1 Introduction

The Solaris threads API allows one thread to stop and re-start another thread by calling the functions:

```
int thr_suspend( thread_t tid );
```

```
int thr_continue( thread_t tid );
```

Threads can also be created in a suspended state by setting the *flags* parameter to *thr_create()*.

POSIX threads does not support the suspend and continue operations on a thread.

6.2 Implementation

To implement *thr_suspend()* and *thr_continue()*, two approaches were considered:

1. Tell the thread scheduler that the specified thread is to be suspended and hence not scheduled to run until further notice.
2. Asynchronously interrupt the to-be-suspended thread.

The first approach is ruled out because it would require changes to the underlying POSIX threads implementation or operating system. Thus an asynchronous mechanism to interrupt another thread is required. The only suitable mechanism available is signals.

It is worth noting that in threaded programs, a signal handler is process wide (i.e. the same for all threads), and the signal mask of blocked signals is thread-specific (i.e. can vary per thread).

In the book *Programming with POSIX Threads* [7], there is an example implementation of the thread-suspend and thread-continue routines, which uses two signals for suspend and continue respectively. This forms the basis of the STL implementation of *thr_suspend()* and *thr_continue()*, with a few modifications.

In essence, STL implements *thr_suspend()* by calling *pthread_kill()* to send a *suspend* signal to the target thread. The signal handler for the *suspend* signal then calls *sigwait()* to block until it receives a *continue* signal. *sigwait()* is one of the few functions that can be safely called from within a signal handler.¹¹

A suspended thread is resumed by another thread calling *thr_continue()*. This function is also implemented by calling *pthread_kill()*, but sends a *continue* signal to the target thread. The signal handler for the *continue*

signal is a null routine. Upon receipt of the *continue* signal the target thread returns from both the *continue* and *suspend* signal handlers to resume whatever it was doing.

By default on Tru64 UNIX, STL uses *SIGUSR1* as the *suspend* signal, and *SIGUSR2* as the *continue* signal. However, the signal numbers used by STL can be changed by setting environment variables: see the *SCL Users Guide* [6].

Solaris documents that *thr_suspend()* does not return until the target thread is suspended. In other words, the thread executing *thr_suspend()* needs to know that the target thread has received the *suspend* signal. (A thread may have temporarily blocked a set of signals, in which case the *suspend* signal is pending until the thread unblocks that signal). Thus within the *suspend* signal handler routine, the thread being suspended needs to indicate to the caller of *thr_suspend()* that it is now suspended. A global semaphore is used for this purpose. The suspended thread calls *sem_post()*, which the thread executing *thr_suspend()* waits upon by calling *sem_wait()*. *sem_post()* is another function that can safely be called from within a signal-handler.

6.3 Further complications

The actual implementation has other factors to consider:

- The use of a global semaphore to acknowledge that a thread is suspended means that STL must serialize access to *thr_suspend()*.

Actually, serializing access to *thr_suspend()* has the advantage that it makes it easier to implement, albeit at the expense of concurrency.

- STL needs to handle a thread suspending or continuing itself, and handle threads that are created in a suspended state.
- STL cannot allow the last non-suspended thread to suspend itself: STL must return EDEADLK in this situation, as documented by Solaris.
- If one thread calls *thr_suspend(X)*, and a fraction later another thread calls *thr_continue(X)*,

¹¹ POSIX 1003.1-1996 defines which functions are entrant with respect to POSIX signals.

STL needs to ensure that the result is that thread X is running, not suspended.¹²

This is achieved by serialising access to *thr_suspend()* and *thr_continue()* by using the same mutex for both functions.

- STL prevents a thread from being suspended whilst it holds one of the internal STL locks (such as a mutex for a join-any thread list) by calling *pthread_sigmask()* to temporarily blocking signals, to prevent the application from hanging within STL.

7 STL Status

As previously stated, STL has been released as part of the Solaris Compatibility Libraries for Tru64 UNIX. STL is also being ported to Linux.

7.1 Linux port

A provisional STL library has been built on Linux, and some test programs run. Problems with thread suspend and continue have been encountered, and are as yet unresolved.¹³

It is hoped that the Linux version of STL will be complete and available by the time of the USENIX Annual Technical Conference in June 2001.

7.1.1 Experiences from the Linux port

The main requirements for porting STL to another platform are that the target platform has an ANSI-C compiler ([ANSI89], ratified by [ISO90]), and a POSIX threads library with the read/write locks extensions.

The GNU C compiler is being used on Linux, with the *-ansi* and *-Wall* switches. This has flagged several warnings, which is to be expected given that STL had never been ported before. The code has been changed and is now more portable.

¹² When a thread processes its pending signals, there is no guarantee that they are processed in the same order that they were received.

¹³ Thread suspend and continue were most troublesome in the original STL implementation on Tru64 UNIX.

When considering the Linux port of STL there was concern about *LinuxThreads* [8], the POSIX threads library on Linux. *LinuxThreads* implements POSIX threads by creating separate processes with the *clone()* system call. But the concern seems unfounded: the only real problem encountered so far with *LinuxThreads* is that threads within the same (logical) process actually have different process identifiers.

Initially it was thought that *LinuxThreads* did not have POSIX threads' read/write locks. The reasons for thinking this were:

- Most POSIX thread functions on Linux have man-pages or info-pages, but the read/write lock functions are not documented.
- A test program built on Linux got unresolved symbols for read/write locks.

The answer, found via the threads programming newsgroup [9], was that when compiling you had to define:

```
_XOPEN_SOURCE=500
```

before including the *<pthread.h>* header-file, in addition to defining:

```
_POSIX_C_SOURCE=199506L
```

These explicit definitions are not necessary on Tru64 UNIX.

Two other problems have been encountered during the Linux port. The first problem was trying to get the shared object library to run an initialization routine. This is achieved by specifying the

```
__attribute__((__constructor__))
```

directive, but you must use *cc* as the link driver, rather than *ld*, for this directive to be recognized. The other problem was with message catalogs, and the *gencat* utility in particular. On Linux *gencat* is white-space sensitive, in accordance with SUSv2. Thus with non-conformant input (that happened to work on Tru64 UNIX), *gencat* on Linux produced blank messages. The solution was to edit the input to *gencat* to be conformant.

7.2 Performance

It is worth restating that STL performance, whilst desirable, has never been a goal: compatibility is the objective. Using STL will always incur some overhead compared to using native POSIX threads.

Table 1 shows how STL affects the performance in a couple of simple tests, where a test was coded in both Solaris threads and POSIX threads. The tests were performed using STL v1.1 on a Tru64 UNIX v5.1 system (a Compaq Alphaserp ES40 with 4 CPUs, but with the number of CPUs active varied from 1 through to 4).

The STL performance numbers look bad in isolation, especially for thread create/join. But these should be considered worst-case figures, and need to be viewed in the context of how frequently each operation occurs in a real application.

Table 1. Relative performance of POSIX threads and STL threads for simple tests.	
<i>Test</i>	<i>Ratio of POSIX:STL performance</i>
Loop of thread create and thread join	varies from 7:1 to 3:1 (depends on # CPUs active)
Loop of mutex-lock and mutex-unlock	1.1:1

For example, for the mutex-locking test loop, just a single thread is executing, so that there is no contention for the mutex. Consequently the fastest path is taken through the POSIX mutex-locking code. In real applications there will probably be some contention for the mutex, which may result in a locking-thread having to do extra work to block on an already-locked mutex, and the unlocking thread also having to do extra work to wake up the blocked thread. This extra work will make the overhead of STL be less apparent.

For the thread-create-and-join loop using STL, a considerable amount of CPU time was observed being spent in system-mode, compared to using native POSIX threads. This indicates a high number of system calls, the most probable cause being the calls to *pthread_sigmask()* to block and restore signals when the thread locks an STL resource.

The way an application uses threads is also a big factor on STL performance. For example, consider how a threaded program might be coded to find the first 10,000 prime numbers, using a 4-CPU system. One approach might be to create a new worker-thread to test if one specific number *N* is prime (for *N* = 1, 2, 3, 4, etc.), and to have three of these worker threads concurrently active (the main thread makes four threads; i.e. one per CPU). But a better approach would be to create three "permanent" worker-threads that loop repeatedly, testing successive numbers for prime. The latter ap-

proach requires more synchronization between the threads to determine which number to test next, (whereas in the former case the main thread can tell each new thread which number to test via the user-argument to the thread-create routine), but it avoids the overhead of repeatedly creating threads. Both programs require synchronization when a prime is found, to increment a global counter and to store the new prime number in a global array.

Table 2 shows the comparative performance of two prime-number programs coded to each approach, using both the native POSIX threads and STL. Values in the table give the number of primes found per second, using the 4-CPU ES40 again. Bear in mind that for half the numbers tested for prime, the test will complete within a very few instructions, because even numbers are not prime.

Table 2. Performance of prime-number programs. Results in primes-per-second: higher is better.			
<i>Program</i>	<i>POSIX threads</i>	<i>STL threads</i>	<i>Ratio POSIX:STL</i>
New thread for every number	2.5K	0.6K	4.3:1
Three permanent worker threads	63K	61K	1.03:1

The results show that for the case when a new thread is created for every number being tested, the overhead of STL's thread-create and thread-join has considerable impact: the STL program is over four times slower. But by creating the worker-threads just once and using additional synchronization, the overall performance is much higher in both cases, and the overhead of STL is minimal.

The results confirm that the overhead of STL is substantial for the areas of thread creation, thread termination, and joining a thread. The results also show that the overhead of STL for synchronization object manipulation is low. It is envisaged that threaded applications will use the synchronization routines much more frequently than the thread routines, and hence the general overhead of STL should be low.

7.3 Future plans

STL is open-source. The hope is that it will be enhanced and extended by anyone who finds it useful.

Completing the Linux port of STL should broaden its potential use.

8 Summary

This paper describes STL, a Solaris-compatible threads library, which layers upon a POSIX threads library. STL enables applications that use the Solaris threads API to be recompiled on another UNIX platform that supports the POSIX threads API.

This paper describes the major functionality of STL, and how it is implemented.

STL is freely available in open-source format as part of SCL, and a binary library is available for Tru64 UNIX.

References

1. Sun Microsystems.
Solaris Multithreaded Programming Guide.
Prentice Hall, 1995. ISBN 0-13-160896-7.
2. Dave Butenhof.
Threaded Programming Standards.
http://csa.compaq.com/CTJ_Article_29.html
February 2000.
3. Bryan O'Sullivan. 1997.
Answers to frequently asked questions for comp.programming.threads.
<http://www.serpentine.com/~bos/threads-faq/>
4. Sun Solaris to Compaq Tru64 UNIX Porting Guide.
http://www.unix.digital.com/faqs/publications/pub_page/porting.html
5. *Solaris Compatibility Libraries (SCL) for Compaq Tru64 UNIX*.
<http://www.tru64unix.compaq.com/complibs/>
6. *Solaris Compatibility Libraries' Users Guide for Tru64 UNIX*.
http://www.tru64unix.compaq.com/complibs/documentation/html/scl_ug.html
7. David R. Butenhof.

Programming with POSIX Threads.

Addison-Wesley, 1997. ISBN 0-201-63392-2.

8. Leroy Xavier.

The LinuxThreads Library.

<http://pauillac.inria.fr/~xleroy/linuxthreads/>

9. *comp.programming.threads*

Internet newsgroup for discussing threads programming.

Standards

ANSI89: American National Standards Institute (ANSI): *American National Standard X3.159-1989 - ANSI C*

ISO90: International Standardization Organization (ISO): ISO/IEC 9899:1990, *Programming languages - C*.

POS90: ISO/IEC standard 9945-1:1990: [IEEE Std 1003.1-1990]. *Information technology - Portable Operating System Interface (POSIX®) - Part 1: System Application Program Interface (API) [C Language]*.

POS96: ISO/IEC standard 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition]. *Information Technology-Portable Operating System Interface (POSIX®)-Part 1: System Application: Program Interface (API) [C Language]*.

UX98: The Open Group. *Single UNIX Specification Version 2*. 1998.

<http://www.unix-systems.org/>

Are Mallocs Free of Fragmentation?

Aniruddha Bohra*

*Department of Computer Science
Rutgers University
Piscataway, NJ 08854
bohra@cs.rutgers.edu*

Eran Gabber

*Lucent Technologies – Bell Labs
600 Mountain Ave.
Murray Hill, NJ 07974
eran@research.bell-labs.com*

Abstract

`Malloc(3)` is considered to be a robust building block. However, we found that many `malloc` implementations suffer from excessive heap fragmentation when used with Hummingbird, a long-running application which stores a large number of fixed-sized and variable-sized objects in dynamic memory. This paper characterizes the dynamic memory activity pattern of Hummingbird and GNU Emacs. It compares the behavior of nine different `mallocs` when used with Hummingbird and GNU Emacs dynamic memory activity traces. In the Hummingbird case, the best `malloc` caused 30.5% fragmentation (increased heap size above the amount of live memory), while the worst `malloc` caused a heap overflow. In the GNU Emacs case, the best `malloc` caused 2.69% fragmentation, and the worst one caused 101.5% fragmentation.

1 Introduction

Dynamic memory allocation is considered to be a solved problem for most applications. The ubiquitous `malloc(3)` routine is a part of the C run-time library, and most programmers use it without a second thought. Much research and development efforts have been invested in optimizing `malloc` to fit the typical allocation pattern of applications, which is characterized by a small number of object sizes [2].

Most real (commercial) applications consider memory allocation performance early on. They often rewrite the memory allocator specifically to tune it

to their application. For example, the Apache web server [8] and a large n -body simulation program [1, column 6] contained specialized implementation of `malloc`.

We also considered `malloc` to be a robust building block, and we used it with Hummingbird [7], a lightweight file system for caching web proxies. To our surprise, when we run Hummingbird on several operating systems with different `malloc` implementations, the heap size of the process was several times larger than the total size of live memory objects. We knew that it was not a memory leak problem, since we run Hummingbird under Purify [5]. When we used other `malloc` implementations, most of them also caused excessive heap fragmentation (increased heap size). This is a serious problem, since it may cause heap overflows, thrashing or reduce the size of memory that can be used to store live objects. One `malloc` implementation even caused a heap overflow, which is unacceptable.

Hummingbird implements a memory-based cache, which stores variable-sized objects in addition to fixed-sized objects. The total size of live objects is fixed. We believe the Hummingbird's memory access pattern is actually quite common for many long-running applications, which store dynamic memory objects of variable sizes. This could be the memory access pattern of many Web related programs that maintain a cache in memory.

Many algorithms for memory allocators have been studied and documented. Most memory allocations are optimized for short-lived programs with a few fixed allocation sizes [9, 2]. Recently, Larson and Krishnan [4] studied the scalability of memory allocators and the impact of memory allocation on long-running applications with fixed-size objects. Not many people have studied the effect

*This work was done when the author was employed at Lucent Technologies – Bell Labs in summer 2000.

of heap fragmentation on long-running applications with variable-sized objects.

While the excessive heap fragmentation was surprising to us, it is not a new problem. System developers have been facing the problem of `malloc` consuming excessive memory due to fragmentation, and have alleviated the problem by implementing their own memory management schemes. For example, the Apache web server [8] uses its own scatter-gather memory allocation scheme.

In this paper we describe the dynamic memory activity pattern of Hummingbird, and compare the operation of multiple `malloc` implementations given the same sequence of memory allocation and deallocation operations, which were captured from a live run of Hummingbird. We observed wide variation of heap consumption and running times between various `malloc` implementations. The best `malloc` we measured was PhK/BSD `malloc` version 42. It caused 30.5% heap fragmentation. The worst `malloc` was SunOS “space efficient” `malloc`, distributed with Sun OS version 5.8. It could not complete the run due to a heap overflow.

Since Hummingbird is not a commonly used program, we looked for a widely available tool that allocates objects of varying sizes and runs for a long time. We picked GNU Emacs version 20.7, and captured its dynamic memory activity when it was used to edit the source files in a large source hierarchy. See Section 3 for details. The memory activity of GNU Emacs is quite different than that of Hummingbird regarding object size distribution, object lifetime, and total amount of live memory, as described in Section 3. In particular, Emacs’s total amount of live memory grows continuously during the run, while Hummingbird’s total amount of live memory is fixed. We measured a smaller variation between the various `malloc`s when used with the Emacs dynamic memory activity trace relative to the Hummingbird trace. The best `malloc` we measured was Doug Lea’s `malloc` version 2.6.6, which caused 2.69% fragmentation. The worst `malloc` was again the SunOS “space efficient” `malloc`, which caused 101.48% fragmentation. PhK/BSD `malloc` was a close fifth with 3.65% fragmentation.

The main contribution of this paper is in exposing an unexpected problem with `malloc`, which is an existing building block that has been extensively optimized. Since the internal algorithms of the various `malloc`s are so different, we did not attempt

to analyze the root cause of this problem. We hope that this paper will help future developers recognize that some `malloc`s may cause excessive fragmentation, which can be alleviated by switching to a different `malloc`. In the long run, we hope that this paper will spur research in dynamic memory allocation in order to analyze and rectify the problem we identified.

The rest of the paper is organized as follows. Section 2 describes the Hummingbird light-weight file system and characterizes its dynamic memory activity. Section 3 describes the GNU Emacs workload and characterizes its dynamic memory activity. Section 4 explains the trace-driven program we used to compare the various `malloc` implementations. Section 5 contains a measurements of the various `malloc` implementations using the Hummingbird and Emacs memory activity traces, and Section 6 concludes.

2 Hummingbird and Its Dynamic Memory Activity

Hummingbird is a light-weight file system for caching web proxies. Caching web proxies are dedicated to caching and delivering web content. Typically, they are located on a firewall or at the point where an Internet Service Provider (ISP) peers with its network access provider. To increase their hit rate, proxies use disks to store large amounts of cacheable objects. Most publicly available caching proxies use the Unix file system to store cacheable object using the Unix file hierarchy. However, the Unix file system is not well-suited for this application, which cause a great performance penalty. Hummingbird is a light-weight portable file-system library that was designed specifically to improve the access time of caching web proxies to cached objects stored on the disk.

2.1 Hummingbird’s Memory Objects

Hummingbird stores two types of objects in main memory: *files* and *clusters*. Files are variable-sized cacheable objects, such as HTML pages and images. Clusters are fixed-sized objects containing files and some meta-data. The caching web proxy provides locality hints to Hummingbird by requesting that

certain files be co-located. These files are usually the HTML page and its embedded images. Hummingbird uses these hints to pack files into clusters. However, the packing occurs as late as possible, which is when space is needed in the main memory. When the contents of a cluster is written to the disk, its associated main memory is freed. Only a small amount of meta-data is left behind in the memory in order to facilitate fast lookup of cached files. Clusters are read into memory and written to disk in one I/O operation to amortize the cost of the I/O. The total size of live objects in Hummingbird is bounded.

Hummingbird maintains three types of meta-data information: file system meta-data, file meta-data, and cluster meta-data. File and cluster meta-data are fixed-sized objects, which are associated with the variable-sized files and clusters. The file system meta-data is needed for the system to maintain state and manage the memory.

In summary, Hummingbird stores fixed-sized and variable-sized objects in memory for various durations (not all objects are short-lived or long-lived). Some memory objects, such as frequently accessed files and clusters may stay in memory for an extended period of time, while other objects, such as files and clusters which are rarely used, stay in memory only for a brief time. Some meta-data objects are never deleted from memory or have very long lifetimes.

2.2 Hummingbird's Dynamic Memory Activity

We instrumented Hummingbird to generate a trace of its dynamic memory activity. The trace we studied was captured from a Hummingbird run corresponding to the processing of an HTTP proxy log of four days. The trace contains about 58 million events (dynamic memory allocations and deallocations). These events correspond to the dynamic memory activity of Hummingbird when it was processing 4.8 million HTTP requests. Those requests contain 14.3 GB of unique data and 27.6 GB total data.

The total amount of live memory in Hummingbird was fixed at about 217 MB. Note that many memory allocation events in the trace have no corresponding deallocation events. These events correspond to the

allocation of memory objects that stayed in memory when the run ended.

Figure 1 depicts the distribution of Hummingbird's object sizes against their frequency. The left portion of the graph, which corresponds to objects less than 128 bytes in size, indicates that there is a large number of fixed-sized objects holding meta-data, and these objects are only of a few fixed sizes. The graph also shows that there is a large number of object sizes longer than 128 bytes. These are the variable-sized objects holding the file data. There are no object sizes which are especially common, except for 32 KB, which is the cluster size. A unique feature of the graph is that the distribution of objects larger than 128 bytes is represented by almost a straight line on the log-log scale. This feature suggests that the distribution of the variable-sized objects is Zipfian [10].

In a Zipf distribution, the frequency of occurrences of an event is inversely proportional to its rank r using the formula $\frac{1}{r^a}$, where a is close to unity. In other words, the relative frequency of the most common event is 1, since its rank is also 1. The relative frequency of the n th most common event is $\frac{1}{n^a}$, since its rank is n .

Table 1 shows the most common Hummingbird object sizes, their relative frequency, and the fraction of the memory allocated to objects of this size. The "total size allocated" column is the total amount of memory allocated for objects of this size (or size range) during the entire run. Note that more than half of the total allocated memory was for objects of size 32 KB, which is the Hummingbird cluster size. Since most disk accesses (reads and writes) are to full clusters, objects containing clusters are created and deleted frequently.

Figures 2 and 3 depict the lifetime of objects. The object lifetime is presented in two metrics: the average amount of new dynamic memory allocated during the lifetime of the object (Figure 2), and the average number of new dynamic memory objects allocated during the lifetime of the object (Figure 3). Both figures show that objects larger than 128 bytes and less than 32 KB are very long-lived, that is, an absence of locality in the patterns of sizes of deallocations and requests for new chunks of memory.

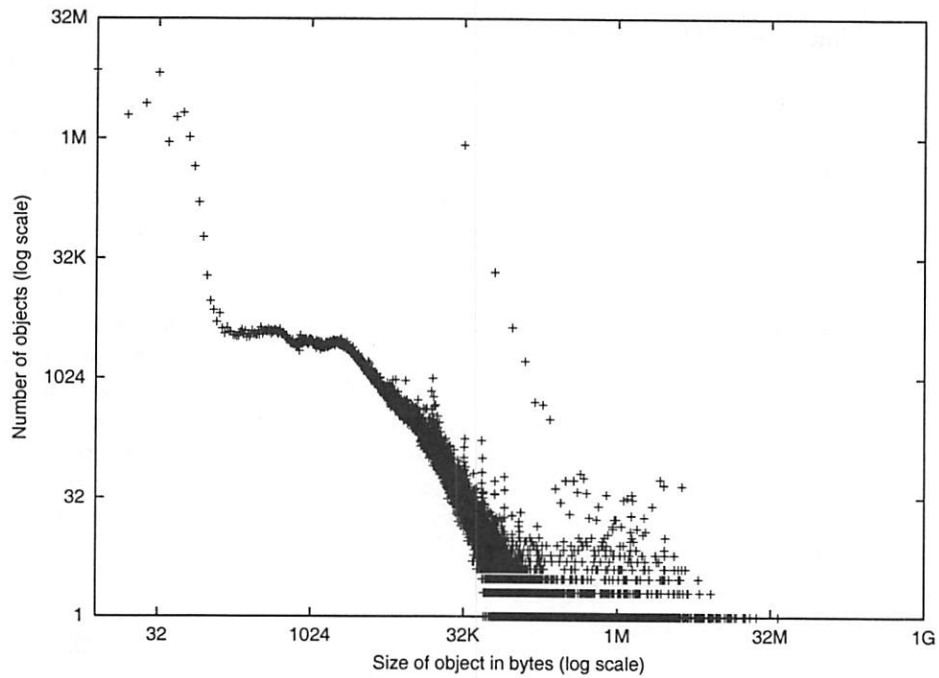


Figure 1: Distribution of Hummingbird's object sizes.

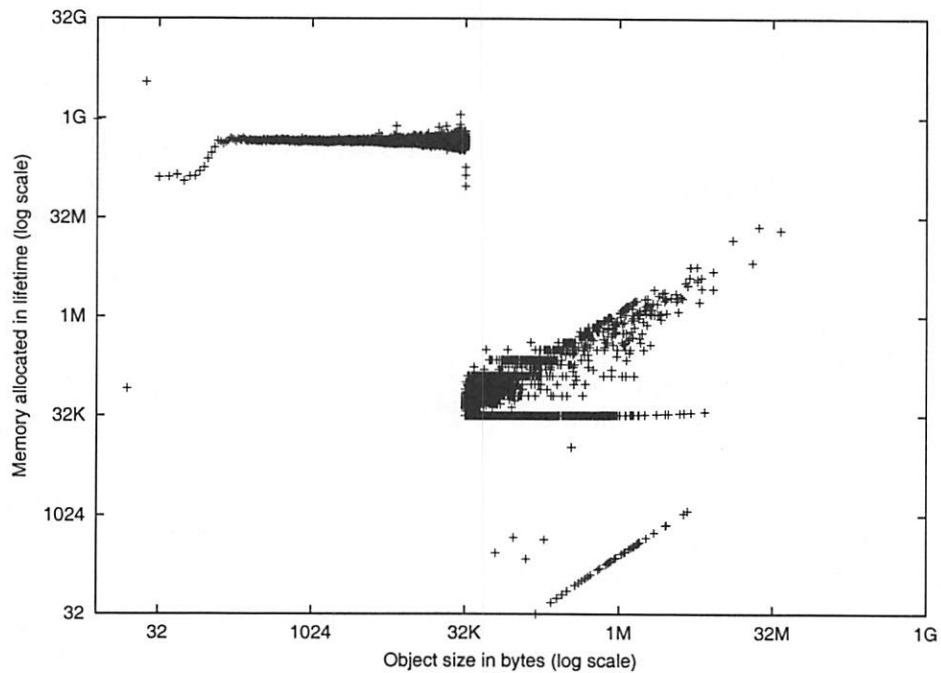


Figure 2: Memory allocated during the lifetime of a Hummingbird memory object as a function of its size.

object size (bytes)	% of total objects allocated	% of total size allocated
8	25.813	0.118
32	23.775	0.436
24	9.848	0.136
56	7.518	0.242
16	7.056	0.065
48	6.604	0.182
64	3.705	0.136
40	3.196	0.073
32768	2.975	55.935
72	1.588	0.066
80-120	0.911	0.045
128-32760	6.647	20.197
32776-39966072	0.364	22.370
total	100.000	100.000

Table 1: The distribution of the most frequent object sizes in Hummingbird. The table is sorted by decreasing frequency. Object sizes are always a multiple of double word (8 bytes).

From the above discussion, we can observe that Hummingbird's dynamic memory activity is very different from the kind of activity that memory allocators are designed and optimized for. Most studies conclude that there are a small number of distinct dynamic object sizes in real programs. Johnstone and Wilson [2] say that 99.9% of the objects are of just 141 sizes! However, Hummingbird allocates a very large number of object sizes (more than 18000). Many mallocs assume that there is a strong temporal correlation between allocation and deallocation sizes. In other words, an allocation is likely to specify the memory size of a recently deallocated object. However, Hummingbird's dynamic memory activity has little correlation between the recently freed objects and new allocation requests.

3 GNU Emacs and Its Dynamic Memory Activity

The second application we studied was GNU Emacs version 20.7. We picked GNU Emacs due to its wide use, and because many people use Emacs for their main working environment, and run a single Emacs session for an extended period of time (many days). GNU Emacs is available from <http://www.gnu.org/software/emacs/emacs.html>.

We instrumented Emacs to generate a trace of its dynamic memory activity. We ran an Emacs macro that listed the contents of `/usr/src/` using the `dired` directory editing mode, visited all of the `*.[ch]` files found, and changed the string `int` to `uint32` in all files. Emacs edited those files sequentially (one at a time). The directory `/usr/src/` contained the source trees of the following Linux kernels: `linux-2.0.34`, `linux-2.1.131`, `linux-2.2.10-siginfo`, `linux-2.2.10-swapmod`, `linux-2.2.10-up-default`, `linux-2.2.12`, `linux-2.2.14`, `linux-2.2.14-mvia`, `linux-2.2.14-up`, `linux-2.2.17`, `linux-2.4.2`, `linux-ctl`, `linux-eager` and `linux-net`. There were 73,212 `*.[ch]` files with total size 2.7GB. Emacs was executing on a PC running `linux-2.2.16-SMP`. The dynamic memory activity trace contained about 20 million memory allocations and deallocations.

Figure 4 depicts the distribution of Emacs's object sizes against their frequency. It shows that Emacs allocated almost entirely small objects (less than 2K bytes). There seems to be two classes of objects based on their allocation frequency: objects that are allocated a large number of times (more than 1,000) and objects that are allocated a small number of times (less than 32). There is no clear correlation between the object size and its allocation frequency.

Table 2 shows the most common Emacs object sizes, their relative frequency, and the fraction of the memory allocated to objects of this size. The "total size allocated" column is the total amount of memory allocated for objects of this size (or size range) during the entire run. Unlike the Hummingbird object size distribution, Emacs has no single object size which dominates the total storage allocated. Moreover, more than 98% of all Emacs objects were small or equal to 648 bytes, and their size was about 65% of the total bytes allocated.

Figures 5 and 6 depict the average number of bytes and objects allocated during the lifetime of an object, respectively. Figure 6 clearly shows that most objects have a similar lifetime, which ranges from 64 K to 1 M object allocations. It seems that those objects were allocated for each source file that Emacs edited, and then they were deallocated when Emacs moved to the next source file.

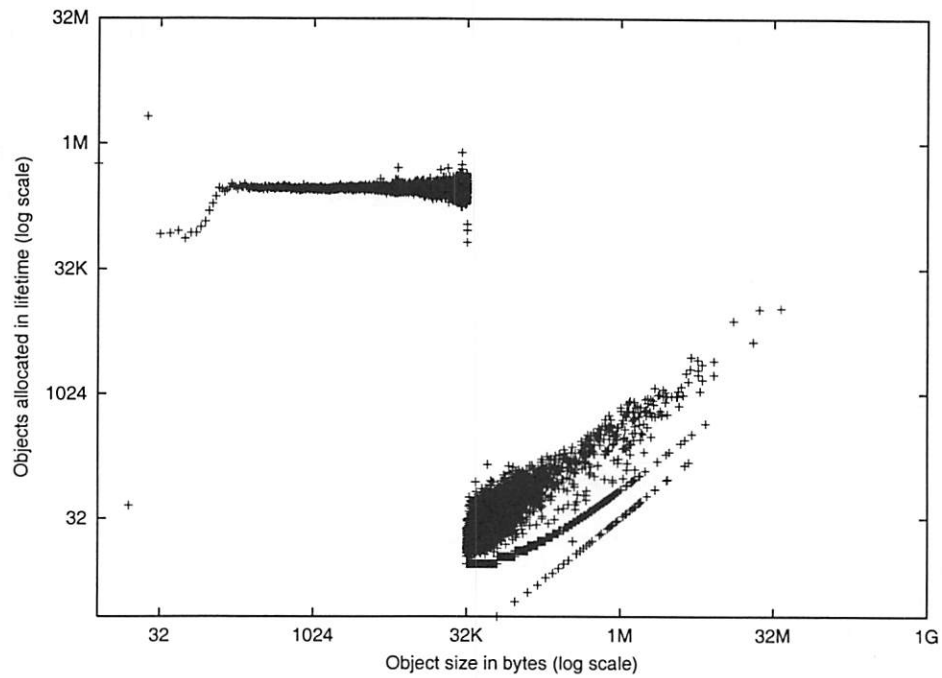


Figure 3: Number of objects allocated during the lifetime of a Hummingbird memory object as a function of its size.

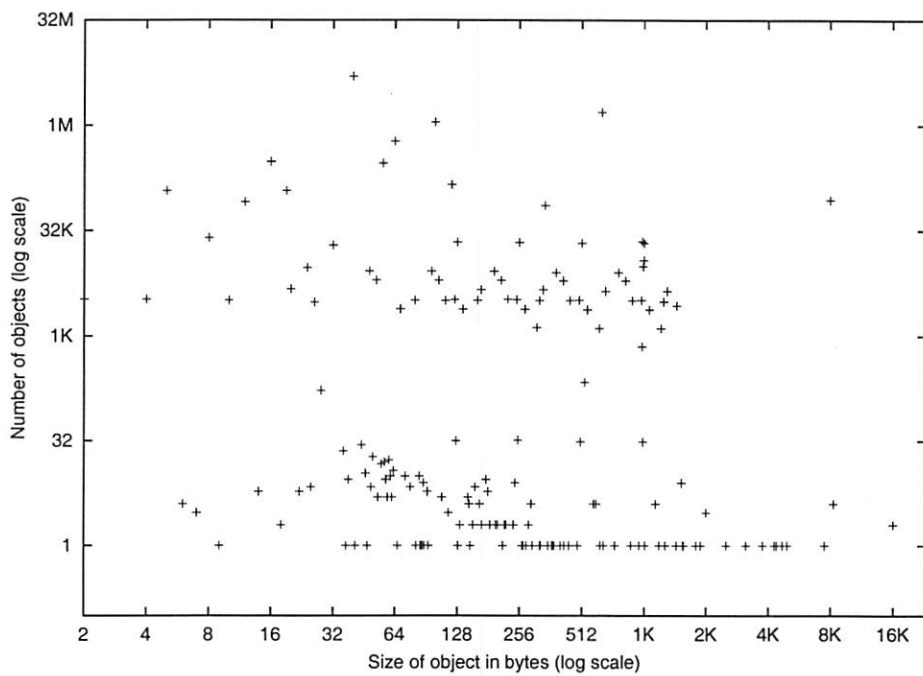


Figure 4: Distribution of Emacs's object sizes.

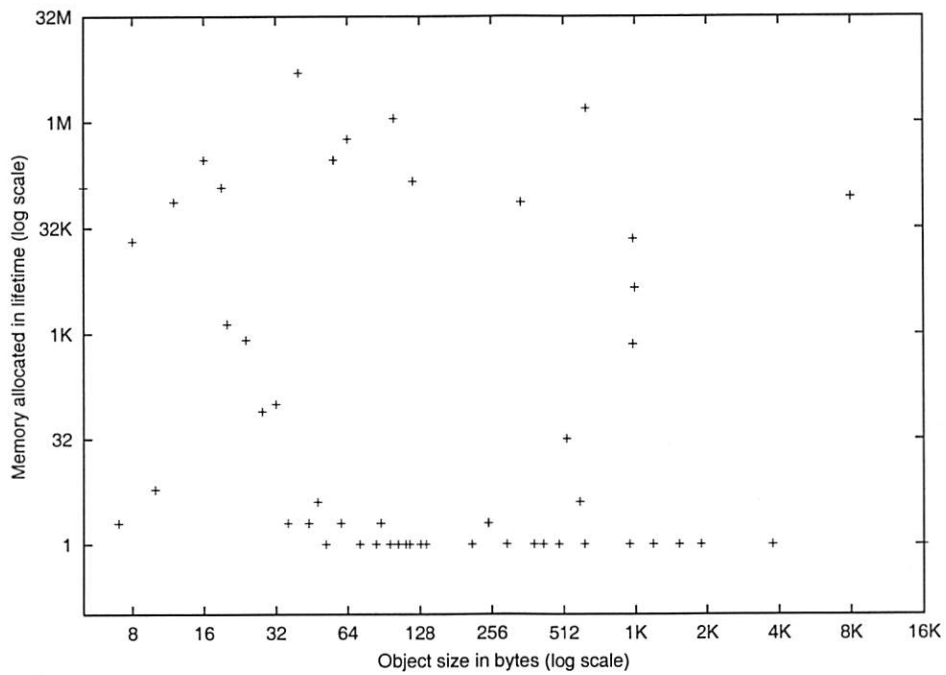


Figure 5: Memory allocated during the lifetime of an Emacs memory object as a function of its size.

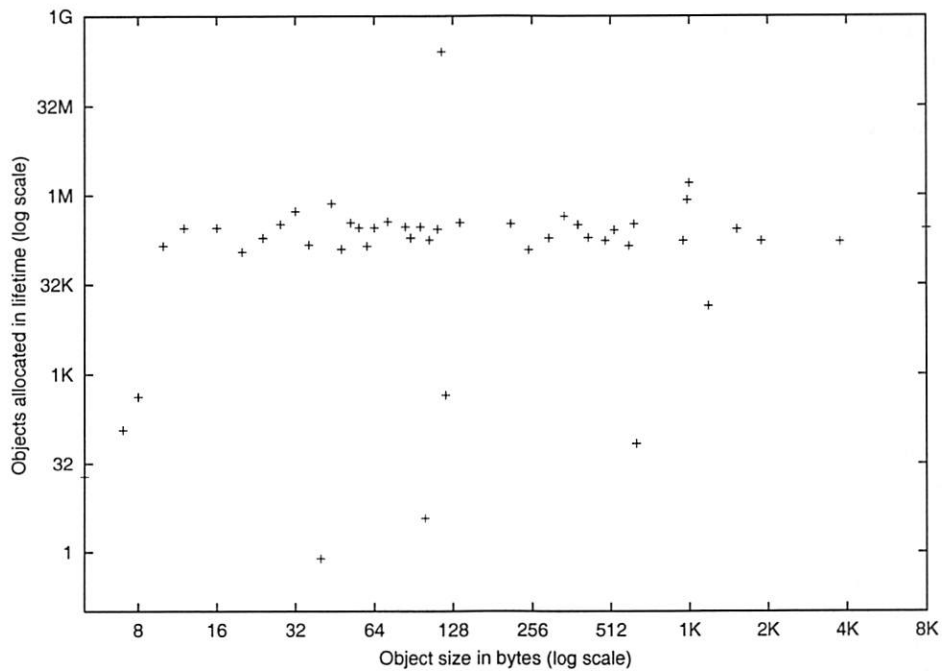


Figure 6: Number of objects allocated during the lifetime of an Emacs memory object as a function of its size.

object size (bytes)	% of total objects allocated	% of total size allocated
40	51.335	8.902
648	15.625	43.897
104	11.368	5.126
64	6.013	1.668
16	3.926	0.272
8	1.496	0.052
16-32	1.549	0.169
48-56	3.075	0.744
72-96	0.141	0.059
112-640	3.554	3.599
656-23568	1.917	35.520
total	100.000	100.000

Table 2: The distribution of the most frequent object sizes in Emacs. The table is sorted by decreasing frequency. Object sizes were rounded to the next multiple of double word (8 bytes).

4 Dynamic Memory Activity Trace

The dynamic memory activity trace was captured from Hummingbird and GNU Emacs runs as described in Sections 2 and 3 above. The Hummingbird trace contained about 58 million events (dynamic memory allocations and deallocations) while the Emacs trace contained about 20 million events.

The memory activity trace is a text file. Each line in the file corresponds to either a memory allocation or deallocation operation. The format of the trace lines is:

```
Allocate <tag> <size>
Free <tag>
```

The *tag* field is used to match the memory allocation with the corresponding memory deallocation operation. We used the virtual address of the allocated area in the instrumented program as the tag. Of course, when we run the trace with different mallocs on different operating systems, the memory allocations will result in different virtual addresses than the tags in the trace file. However, the tags can still be used to match the malloc operations with the corresponding free operations.

An alternate implementation of the trace file might have used the allocation sequence number as the

tag. In other words, the first allocated memory object will get tag # 1, the second will get tag # 2, etc. We did not implement this option since it necessitates auxiliary data structures in the trace generation routines, which may perturb the measured application. However, it is easy to generate this alternate trace format by post-processing the current trace file format.

We wrote a simple driver program that reads the trace and calls the corresponding malloc and free based on the current trace file entry. The driver program keeps a hash table with the tags of all live memory objects. In this way, it can locate the corresponding memory object for the free operation given its tag.

5 Measurements

We measured the heap size of the following nine mallocs when used with the same Hummingbird and Emacs dynamic memory access traces, which were described in Sections 2 and 3 above. We picked mostly open source malloc packages which are common on Linux and FreeBSD. We also included two Solaris mallocs, since the Solaris 3X malloc caused the heap overflow that prompted us to investigate the fragmentation problem in the first place.

The following description of the mallocs is sorted by increasing heap fragmentation at the end of the Hummingbird trace run. This is also the order of entries in Table 3. The fragmentation percentage is computed by $100 \times (\frac{\text{heapsize}}{\text{live memory}} - 1)$.

- **PhK/BSD malloc version 42**

Written by Poul-Henning Kamp [3] and distributed with FreeBSD, NetBSD and OpenBSD. Available from <ftp://ftp.FreeBSD.org/pub/FreeBSD/src/lib/libc/stdlib/malloc.c>.

- **Solaris default**

This is the default malloc distributed on SunOS 5.6.

- **GNU malloc last modified in 1995**

Written by Mike Heartel and distributed with the GNU libraries. Available from <ftp://www.leo.org/pub/comp/os/unix/gnu>.

The latest version of GNU malloc is available from <ftp://ftp.leo.org/pub/comp/os/unix/gnu/malloc.tar.gz>.

- **Modified binary buddy**

A modified binary buddy algorithm, which coalesces any two neighboring free blocks of the same size, even if the resulting block is not aligned on the new block size boundary. This algorithm uses a hash table to determine quickly if a neighboring block of the same size is completely free. This routine was written by the second author of this paper and will be available from <http://www.bell-labs.com/~eran/malloc>.

- **Doug Lea's malloc version 2.6.6**

This malloc is optimized both for speed and fragmentation and is the basis for the GNU g++ malloc. Written by Doug Lea and available from <http://gee.cs.oswego.edu/pub/misc/malloc.c>.

- **Quick Fit malloc**

An implementation of Weinstock and Wulf's fast segregated-storage algorithm based on an array of free lists. available from <ftp://ftp.cs.colorado.edu/pub/cs/misc/qf.c>.

- **CSRI malloc version 1.18**

Written by Mark Moraes from the University of Toronto. Available from <ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz>.

- **Vmalloc written on 1/16/1994**

Kiem Phong-Vo's malloc with the "default" setting, which is optimized for "typical" workloads. The results of the "best" setting were very similar to the "default" setting for both Hummingbird and Emacs traces, so we reported only the results of the "default" setting. Available from <http://portal.research.bell-labs.com/orgs/ssr/book/reuse/license/packages/95/vmalloc.html>.

- **Solaris 3X**

This is a "space efficient" malloc distributed with SunOS 5.8. It is available through linking with `-lmalloc`. Its manual page is `man 3x malloc`.

We ran all tests a dual processor Intel Pentium III with 700 MHz clock speed running SunOS 5.8 (Solaris 8 distribution). Note that the driver program is single threaded, so it did not use the 2nd processor.

5.1 Hummingbird Measurements

Table 3 shows the final heap size and the heap fragmentation at the end of the Hummingbird trace. It also shows the CPU consumption for running the malloc on the full trace. The execution time column in Table 3 indicates that increased CPU consumption does not correspond to reduction in fragmentation. The best malloc is not the slowest, and the fastest malloc does not cause most fragmentation.

Figure 7 shows the heap size of the same mallocs as a function of the time. The rightmost point in all graphs in Figure 7 is the final heap size shown in Table 3. Note that the fragmentation described in Table 3 is independent of the operating system we used. Executing the same malloc code on a different operating system will cause similar fragmentation.

Table 3 shows that the heap fragmentation ranges from 30.5% to infinity (heap overflow). Moreover, most mallocs do not reuse memory properly, which is indicated by continuously increasing heap size in Figure 7.

PhK/BSD performed best among all the studied mallocs for the Hummingbird trace. Not only it had the smallest heap fragmentation, it also did a better job at reclaiming freed areas. Figure 8 compares the heap size of the four best mallocs: PhK/BSD, Solaris default, GNU and modified bin buddy with the live memory. Figure 8 has the same time scale as Figure 7.

Figure 8 indicates that the Solaris default, GNU and modified bin buddy mallocs allocated some area on the heap, and they were not able (or not designed to) reduce the heap size if there was an opportunity to do so. Such an opportunity arises when the last memory area in the heap is freed. Only PhK/BSD was able to reduce the heap size on these occasions, which may explain its overall better operation.

5.2 Emacs Measurements

Table 4 shows the final heap size and the heap fragmentation at the end of the Emacs trace. It also shows the CPU consumption for running the malloc on the full trace.

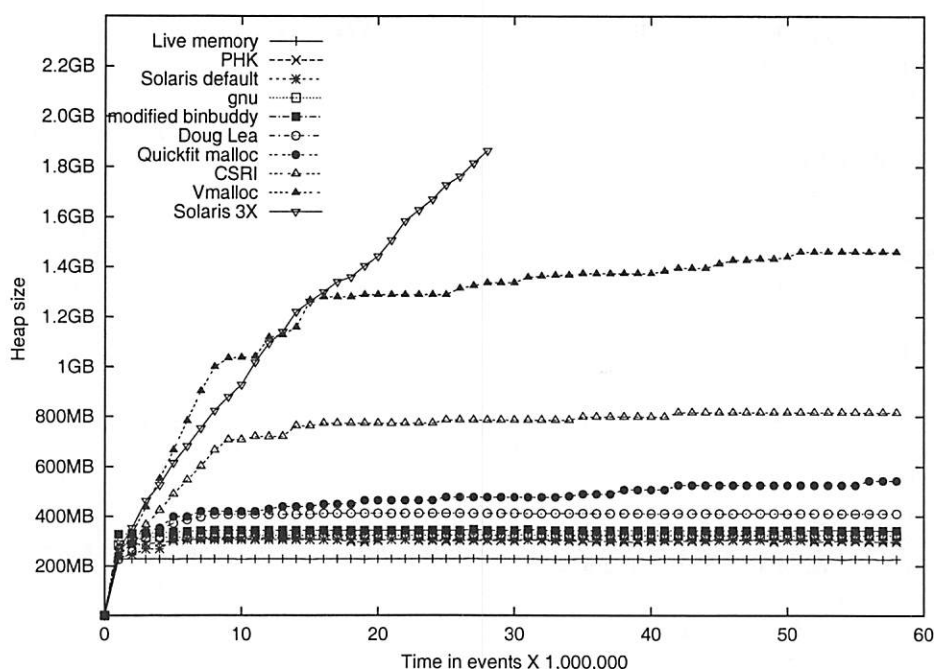


Figure 7: Comparison of the heap size of nine mallocs when used with the Hummingbird dynamic memory activity trace. The *Live memory* line shows the actual live memory size.

malloc package name	final heap size (MB)	% fragmentation	user time (sec.)	system time (sec.)
PhK/BSD	283.8	30.5	448	11
Solaris default	291.7	34.7	360	9
GNU	308.3	41.8	455	17
Modified bin buddy	327.8	50.7	557	11
DougLea	392.6	80.6	641	21
Quickfit	518.9	138.7	457	12
CSRI	778.5	258.8	14171	29
Vmalloc	1364.7	527.7	384	52
Solaris 3X	heap overflow	—	—	—

Table 3: Comparison of the heap size, fragmentation and CPU consumption at the end of the Hummingbird trace. The table is sorted by increasing fragmentation. Live memory at the end of the trace was 217.4 MB.

malloc package name	final heap size (MB)	% fragmentation	user time (sec.)	system time (sec.)
DougLea	136.48	2.69	74	4
CSRI	137.01	3.08	585	4
Quickfit	137.02	3.09	74	3
Vmalloc	137.36	3.35	75	4
PhK/BSD	137.76	3.65	78	4
Solaris default	137.76	3.65	76	4
GNU	161.41	21.44	78	5
Modified bin buddy	172.54	29.82	84	4
Solaris 3X	267.79	101.48	372	8

Table 4: Comparison of the heap size, fragmentation and CPU consumption at the end of the Emacs trace. The table is sorted by increasing fragmentation. Live memory at the end of the trace was 132.9 MB.

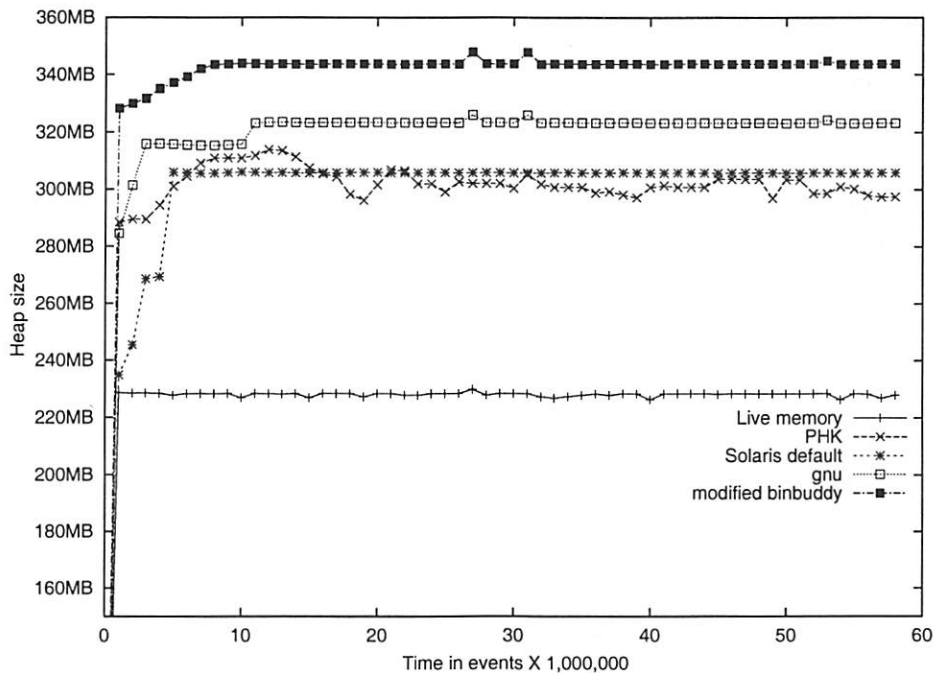


Figure 8: Comparison of the four best mallocs when used with the Hummingbird dynamic memory activity trace. Note that PhK/BSD was able to reduce the heap size on several occasions.

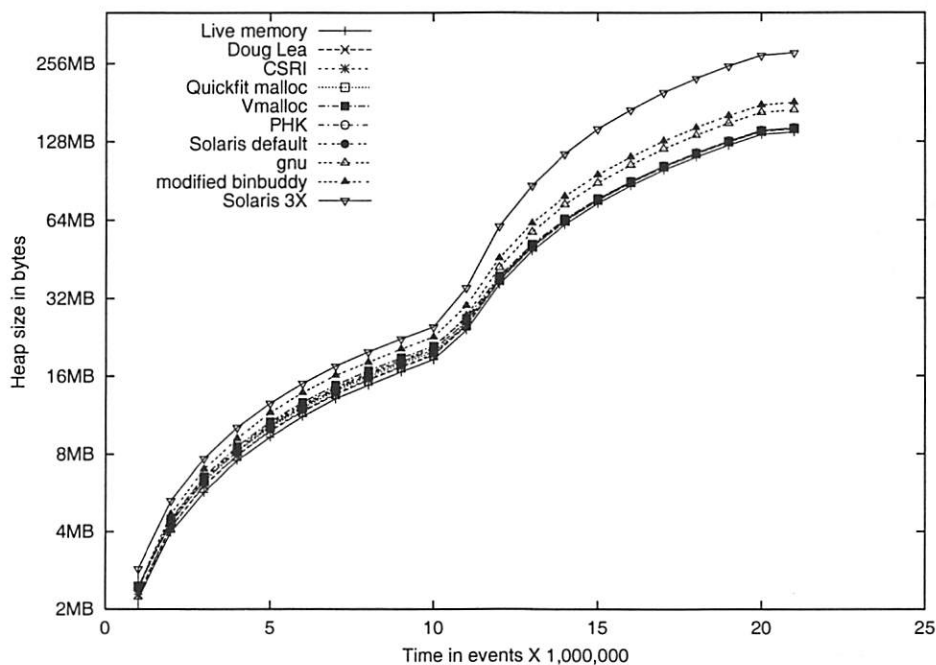


Figure 9: Comparison of the heap size of nine mallocs when used with the Emacs dynamic memory activity trace. The *Live memory* line shows the actual live memory size.

Figure 9 shows the heap size of the nine mallocs as a function of the time. The rightmost point in all graphs in Figure 9 is the final heap size shown in Table 4. Note that most mallocs except GNU, modified binary buddy and Solaris 3X caused very little fragmentation and are very similar to each other in that respect. The Solaris 3X again caused most fragmentation, which may indicate that its implementation is broken.

5.3 Discussion

We did not study the internal algorithm of any of the mallocs we measured, except for the modified binary buddy algorithm. We treated them as “black boxes”, since this is the way most application developers use them. Thus we can not explain why some mallocs consumed much more heap space than others.

One could argue that we can resolve the memory fragmentation problem in Hummingbird by rounding the sizes of all memory blocks to the next power of two, and then use an existing malloc. In this way, splitting memory blocks and coalescing memory blocks will not cause fragmentation. However, as Table 3 shows, the modified binary buddy allocator caused 50.7% fragmentation, which is much worse than the best allocator, which caused only 30.5% fragmentation. Remember that the binary buddy actually allocates memory only in chunk sizes which are a power of two. Moreover, some implementation of malloc append a header to all memory blocks. Thus coalescing two blocks of the same size will generate a block whose size is slightly larger than twice the size of each original block. Thus this method will not eliminate fragmentation.

The large increase of the heap size experienced with Solaris 3X malloc when used with the Hummingbird trace is within the theoretical bounds for worst case fragmentation of first fit and best fit allocation algorithms [6]. The maximal memory needed for first fit is about $M \log_2 n$, where M is the total size of live memory, and n is the size of the largest object. The maximal memory needed for best fit is about Mn , which is much larger. In our case, M is 217.4 MB and n is 39.9 MB. Considering the above worst case analysis, most mallocs (except Solaris 3X) required a much smaller amount of memory than the worst case.

The Emacs measurements showed that even when the object distribution is “more typical”, some mallocs are better than others. In particular, the GNU malloc caused 21.4% fragmentation, while the better mallocs caused about 3% fragmentation.

6 Summary and Conclusions

We studied the behavior of Hummingbird, a long-running program with dynamic memory allocation and deallocation patterns which do not conform to the typical pattern for which dynamic memory allocators are optimized. We also studied the dynamic memory activity of GNU Emacs when it was used to edit files in a large source hierarchy. We studied the fragmentation caused by the different implementations of malloc when used with the same Hummingbird and Emacs dynamic memory activity traces. We found that most mallocs caused extensive fragmentation for the Hummingbird trace. However we also found that some of them performed well. This is in contrast with the Emacs trace, which caused little fragmentation for most mallocs. However, there was a noticeable difference between the best mallocs and the rest. No malloc performed the best for both traces, while one malloc was consistently the worst.

While dynamic memory management for programs that allocate a small number of object sizes has been studied extensively, further research is needed to understand the dynamic memory management for long-running programs which allocate a large number of memory object sizes with varying sizes and lifetimes. Moreover, the low memory fragmentation of the best malloc for Hummingbird, PhK/BSD malloc [3], is purely serendipitous based on a correspondence with its author, Poul-Henning Kamp. Moreover, P-H Kamp did not claim to have tried to reduce fragmentation in case of a very large number of object sizes. In other words, he does not know why it performed so well on our workload.

We hope that this paper will spur renewed interest in dynamic memory allocation, and would lead to better understanding why particular dynamic memory allocation schemes work better for the kind of dynamic memory activity described in this paper. Future malloc implementors should consider a dynamic memory activity pattern similar to Hummingbird’s when updating their code. At the min-

imum, application developers should become aware of the excessive memory fragmentation problem described in this paper, and if they encounter one, they should try to alleviate it by picking a different malloc package.

7 Availability

The Hummingbird and Emacs memory activity traces, the source of our driver program, and the source of the modified bin buddy allocator will be available at <http://www.bell-labs.com/~eran/malloc/>.

In addition, Benjamin Zorn has a site with links to several mallocs: <http://www.cs.colorado.edu/~zorn/Malloc.html>.

Acknowledgements

The authors would like to thank the anonymous referees and our FREENIX shepherd, Alan Nemeth, for their invaluable comments. The authors would also like to thank Emery Berger for his comments, and Michael Flaster for his Emacs wizardry.

References

- [1] Jon Bentley. *Programming Pearls, 2nd Edition*. Addison Wesley, 2000.
- [2] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the Intl. Symp. on Memory Management (ISMM)*, pages 26–36, Vancouver, Canada, October 17–19 1998.
- [3] Poul-Henning Kamp. Malloc(3) revisited. In *Usenix 1998 Annual Technical Conference: Invited Talks and Freenix Track*, pages 193–198. Usenix, June 1998.
- [4] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the Intl. Symp. on Memory Management (ISMM)*, pages 176–185, Vancouver, Canada, October 17–19 1998.
- [5] Purify. Rational Purify. <http://www.rational.com/products/pqc/index.jsp>.
- [6] J. M. Robson. Worst case fragmentation of the first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, August 1977.
- [7] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher Stein. Storage management for web proxies. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25–30 2001.
- [8] Apache Webserver. Apache software foundation. <http://www.apache.org>.
- [9] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management (IWMM 1995)*, pages 1–116, Kinross, Scotland, UK, September 1995. Springer Verlag LNCS 986.
- [10] G. K. Zipf. *Human Behaviour and the Principle of Least Effort*. Cambridge Press, 1949.

Sandboxing Applications

Vassilis Prevelakis

vp@prevelakis.net

*Department of Computer and Information
Science*

University of Pennsylvania

Diomidis Spinellis

dds@aueb.gr

*Department of Technology and
Management*

*Athens University of Economics and
Business*

Abstract

Users frequently have to choose between functionality and security. When running popular Web browsers or email clients, they frequently find themselves turning off features such as JavaScript, only to switch them back on in order to view a certain site or read a particular message. Users of Unix (or similar) systems can construct a sandbox where such programs execute in a restricted environment. Creating such a sandbox is not trivial; one has to determine what files or services to place within the sandbox to facilitate the execution of the application. In this paper we describe a portable system that tracks the file requests made by applications creating an access log. The same system can then use the access log as a template to regulate file access requests made by sandboxed applications. We present an example of how this system was used to place Netscape Navigator in a sandbox.

1. Introduction

The sad truth is that whichever operating system one may be using, running large monolithic programs is a security risk. The original Unix philosophy of having simple dedicated tools that could be combined to carry out complex tasks is being abandoned. Instead, huge programs such as Netscape Communicator and the Star Office suite have been constructed by simply piling up one marginally useless feature after another. Nevertheless, users like fancy features and will use such programs despite our many philosophical and stylistic objections. Therefore, our rear guard action must be to alleviate the detrimental effects the use of such programs may have on the security posture of our system.

The concept of creating restricted environments for programs that are considered unsafe is by no means new. Unix itself offers many restrictions to what processes can and cannot do. Where additional security is required (e.g. by the `ftp` daemon) the `chroot(2)` system call is used to restrict access to a specific area of the filesystem. In this paper we examine how these access restrictions can be used to create a safe execution environment, and describe a tool that supports the construction and operation of such sandbox-type environments. We illustrate our

approach by sandboxing Netscape Communicator and discuss the wider implications of the use of such mechanisms.

2. Access Restrictions

Let us consider the classic dilemma. A typical user receives a Microsoft Word file and would like to see what is inside but is afraid of what will happen to his workstation if the file contains a virus. One solution would be to place the program (Word) along with the suspect file in a controlled environment (which from now on we will refer to as sandbox) and open it there. If the file is infected, the effects of the virus will be localized, but not entirely eliminated as we shall see later on.

The sandbox must contain all the files needed for executing the application. Gathering a list of these files is a non trivial task. Applications depend in utilities such as `lpr` or `mail`, and on shared libraries, loadable modules, configuration files, and the operating system files required by various C library functions (IP service names, localized messages, time zone specifications, etc.). A key consideration is to make sure that the program does not escape from the sandbox, and that the sandbox system never assigns the program greater privileges than it would have if it ran outside the sandbox. Thus, the sandbox must disable access to `setuid` programs, or not allow them to be executed with permissions other than the ones

This work was supported by DARPA under contract F39502-99-1-0512-MOD P0001.

given to the user running the application. Moreover, to prevent the application from gathering information about the system by accessing files such as `/etc/passwd` and `/etc/hosts` we need to substitute them with “sanitized” versions that contain just the information that the application requires to perform its tasks.

More sophisticated systems like Janus [1] support centralized policies with fine grained control over the resources that the process may use (e.g. file descriptors, memory, file system space etc.). Our approach dynamically evaluates the requirements of a given program, creating a sandbox specification that will not affect its operation. In doing so we strive to balance what we would like to control against the effort in specifying these restrictions.

3. Approach Overview

Our target audience is the typical Linux or *BSD user, i.e. people with their own PC where they have root access, but are not Unix gurus or security specialists. This orientation has strongly influenced our approach. Our sandbox is based on system-provided services such as `chroot` and `mount`. Access to these system calls has important implications regarding system security. However, since our user already has root access we need not worry about the user abusing these calls and can concentrate on automating the generation of secure sandboxes.

Based on this premise, our approach for sandboxing applications is based on the following steps:

1. Run the application with known benign input to create access logs specifying its file access behavior.
2. Use the logs to create an access list that can be morphed into a typical `chroot` environment (like the one used by `ftpd`). The user can augment this list based on the particular requirements of the application.
3. Create the sandbox as a `chroot` environment based on the access list.
4. Run the application with untrusted input in the sandboxed environment thus denying it access to unauthorized files.

Nowadays, operating system releases every few months are commonplace. This places an enormous burden on the developers of software that requires “special” access to the operating system (e.g. kernel

modules). While external kernel interfaces (system calls, `ioctl`s etc.) evolve slowly, internal kernel data structures and interfaces are more volatile. Furthermore, subtle but annoying differences between the various systems make the support of kernel based programs a full time job. We, therefore, decided to stay in user land and invest our resources (time) on making the system portable and flexible.

4. The FMAC Tool

To support the approach we outlined, we designed the File Monitoring and Access Control (FMAC) tool. The tool implements a filesystem that mirrors the system’s existing file structure. With the FMAC filesystem mounted on the workstation, applications are run with a `chroot` operation that limits their access to the FMAC managed filesystem. The FMAC tool supports two modes of operation:

passive whereby FMAC logs file requests while allowing them to go through, and

active when FMAC only honors file access requests that are authorized by a user-specified access list.

Initially applications are run with FMAC in passive mode to create the access log (step 1 of our approach). After the sandbox specification has been created (step 2) FMAC is run in active mode (step 3) and applications can be executed with untrusted input in the FMAC `chroot` environment (step 4).

We implemented two versions of the FMAC tool: one based on a user-level NFS server and one on a Perl filesystem [2]. The user-level NFS server is efficient and highly portable. It is available on most *BSD and Linux distributions. The Perl filesystem trades runtime efficiency for flexibility. It runs without any modifications on any platform supporting Perl filesystems (currently Linux on Alpha and Intel CPUs). In the following paragraphs we describe the two FMAC implementations.

4.1 FMAC as an NFS Server

The FMAC NFS server runs as a user-level process without the need for NFS server support to be present in the kernel. The FMAC filesystem is mounted by the standard NFS client, so the system must be able to mount filesystems using the NFS protocols.

The FMAC tool uses a different port from the well-known NFS port, so that it can coexist with a standard NFS server. All requests from the FMAC

filesystem are processed by the tool which performs a lookup in the access list in order to determine its response to the request. If the filename is found in the access list, then the permissions reported by the FMAC server are constructed by performing a logical AND operation between the permissions in the access list and the those in the underlying filesystem. If the filename is not present in the access list, the FMAC server reports that the file does not exist.

Normally NFS does not allow requests to cross filesystems (i.e. if you export two filesystems `/` and `/usr`, a client mounting only `/` will not be able to access files on `/usr` unless this filesystem is mounted as well). The reason for this limitation is that inodes/vnodes are guaranteed to be unique only within a single filesystem.

In our system we want to be able to view the entire local file hierarchy as a single filesystem so that there is no need for multiple mount points within the `chrooted` environment.

We, therefore, modified the NFS server to allocate file handles dynamically and maintain a lookup table. The implication is that the NFS server is no longer stateless. This is contrary to the NFS philosophy of delegating state to the client, but in our case we felt that our decision was acceptable because:

- This is not a general-purpose NFS server but an application that is intended to run on the same machine as the sandboxed application. If the machine crashes so will the client.
- This only affects the passive mode when we have to construct the file access list in memory. In the active mode the access list is retrieved from a file. In this case the FMAC server may be restarted without disturbing the client.

4.2 FMAC as a Perl Filesystem

The Perl filesystem (PerlFS) is a combination of a Linux kernel module and a Perl extension that make it possible to write filesystem implementations in Perl instead of C. Perl filesystems are object classes conforming to the PerlFS interface. Compared to typical filesystem implementations written in C, Perl filesystems are less dependent on the underlying operating system implementation. The PerlFS interface abstraction isolates the filesystem implementation from operating system changes; the same code will run on all systems supporting PerlFS.

Like the NFS-server implementation, the Perl filesystem allows multiple partitions to be mounted under the same directory hierarchy and dynamically

allocates unique inode numbers for existing files. Two Perl associative arrays are conveniently used to map inode numbers to file names and vice versa. To avoid name aliasing problems created by hard links across files, and the multiple ways a directory hierarchy can be traversed to reach a file, the native filesystem `stat(2)` system call is used to obtain the original unique device/inode number pair as a basis for creating the dynamic inode number.

The high-level interface of the Perl filesystem—trading runtime efficiency for flexibility—allowed us to implement the FMAC tool in less than 2000 lines of Perl code. We utilized Perl's excellent support for regular expressions to experiment with different ways to specify the file access request list. The author of the Perl filesystem module advertises it as an alpha version. However, after some recent improvements that added support for the `mmap` functionality needed to load executable files and shared libraries, we were able to run a number of programs in the `chrooted` environment without a problem.

5. Example

Netscape Communicator is a unified web browser and email client (among other things). Its use creates enormous security and privacy concerns since it has full access to the files of the user. Moreover, Netscape Communicator maintains files such as `bookmarks.html` and `cookies` that may provide hints about the browsing habits of the user. Our objective is to be able to run this program in a stripped down environment where:

- It will have no access to the user files.
- It will have access only to special sanitized versions of the system files.
- The user will be able to create temporary “new” installations of the Netscape user directories for accessing suspicious sites.
- The program will be able to function as a browser and a `pop/imap` client.

5.1 Methodology

We ran Netscape Communicator version 4.75 with the FMAC system in passive mode and we extracted, using FMAC in passive mode, the file hierarchy shown in Figure 1.

We separated the files into two categories, *system* files that are common to all installations (e.g. the `/netscape` hierarchy) and *user* files that are personal files accessible by the user. The former

category is typically read-only and consists of files that are not owned by the user. These are the files that will be handled by the access control part of FMAC as we will see later on.

```

/bin/
  /sh
/dev/
  ...
/etc/
  /group
  /kerberosIV
    /krb.conf
  /localtime
  /login.conf
  /networks
  /pwd.db
  /resolv.conf
  /spwd.db
/usr/
  /X11R6
    /bin
    /lib
      /X11
        /XKeysymDB
        /app-defaults
        /locale
          ...
  /bin
  /su
  /lib
    ...
  /libexec
    /ld.so
  /local
    /netscape
      /netscape
  /share
    /zoneinfo
      ...
/users
  /bob
    ...
/var/
  /X11
    /app-defaults
  /db
    /kvm_bsd.db
  /mail
    /bob
  /run
    /ld.so.hints

```

Figure 1: Files required by the Netscape Communicator sandbox

For most files in the system category we simply need to ensure that access is read only. However a few of them deserve special attention. For example the `/etc/passwd`, `/etc/master.passwd` (or the hashed versions of them, `pwd.db` and `spwd.db`) and `/etc/group` should not be accessed directly since they probably contain information that we would like to keep out of the sandbox. Such information may include user names, group assignments and most

importantly passwords. Another consideration is that there should be no `setuid` program in the sandbox.

The consequence of the above is that the list of files produced by the analysis phase must be examined to determine which files need to be replaced by sanitized versions.

Once the sandbox is built, we can execute the program using a command like:

```

chroot /users/bob/sandbox \
  /bin/su bob -c \
  /usr/local/netscape/netscape

```

Notice that we use the `su` program in the sandbox to reduce the process privileges to those of user `bob`. Since the `suid` bits are ignored within the sandbox, processes within the sandbox cannot use `su` to become root.

5.2 Sandboxing using only `chroot`

If we do not wish to use the FMAC tool for the active phase, we will need to create an actual file hierarchy by copying all the files included in Figure 1 to another part of the filesystem. We would then run the `chroot` command as in the earlier example.

This approach relies only on the standard system tools for the active phase. It also allows the sandbox environment to be moved to other similar platforms with little or no customization. Thus, in a company environment we can create the Netscape sandbox in one machine and then copy it to the machines of all the other employees.

However, copying all these files is wasteful and more importantly, we will need to keep track of all the duplicates and update them every time we upgrade the application, or the operating system. Employing links from within the sandbox to the actual files is quite difficult. Symbolic links cannot be used, while hard links require that the linked files are in the same filesystem and may only be used to link files, not directories. Also the extensive use of hard links, may lead to confusion.

5.3 Sandboxing using FMAC

The FMAC system creates a virtual filesystem by transparently providing access to the real files while at the time enforcing access controls on the basis of a user supplied access list.

To use FMAC we have to start the modified NFS server or the Perl filesystem module and mount the FMAC filesystem, inside the sandbox before closing the sandbox.

To facilitate the configuration of the FMAC system, the list of accessed file produced during the passive phase is the same has the same format as the ACL file used in the active phase.

The ACL contains one line for each file or directory. Its format is as follows:

<permissions> <path> [<actual path>]

permissions is a string that denotes the access permissions allowed for the file.

path is the path to the file requested by the sandboxed application,

actual path is an optional argument that allows us to provide a substitute for the requested file. Only files may be substituted.

For example Figure 2 contains an extract from the Netscape ACL:

```

_X_    /usr/lib
R_     /var/run/ld.so.hints
_X_    /var/run
_X_    /var
R_X_   /usr/libexec/ld.so
_X_    /usr/libexec
R_X_   /usr/bin/su
_X_    /usr/bin
_X_    /usr
R_     /etc/pwd.db  /var/tmp/pwd.db
R_     /etc/spwd.db /var/tmp/spwd.db
. . .

```

Figure 2: Structure of Communicator ACL file.

The permissions apply only to the user (i.e. the traditional Unix “group” and “others” permissions are gone) because only the user will be accessing the files from within the sandbox. The permissions that may be specified are, read, write, execute (access for directories) and create (that allows the named file to be created if it does not exist).

Moreover, the sandbox does not grant additional privileges to the user – the sandbox permissions are *in addition* to the existing Unix permissions that apply to the files.

In the last two lines in Figure 2, we provide substitutes for the password database files (pwd.db and spwd.db).

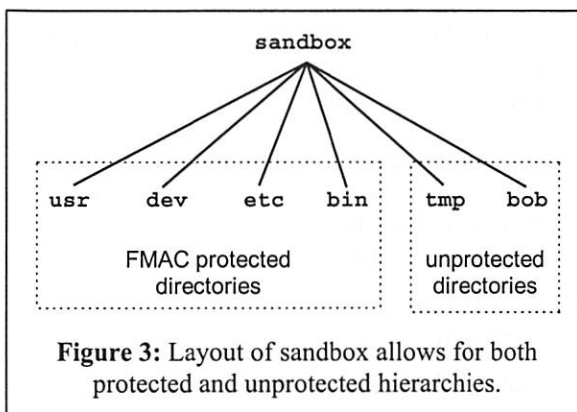
5.4 Constructing the ACL

When running the application in passive mode we need to exercise as much of its functionality as we

can. For example, not accessing the help files will mean that they will be left out of the ACL.

Clearly, we can manually edit the ACL afterwards and correct some obvious omissions, but its is usually easier to have the FMAC system prepare it for us.

A more serious question is how to treat directories that are going to have extensive file creation, such as the Web page cache directories of the Communicator. We did not want to clutter the ACL with definitions for these files, so we decided to provide some directories inside the chrooted environment that are not controlled by the FMAC system. We created the hierarchy shown in Figure 3



The directory containing the user files (/bob) is not the real home directory of the user; it is an empty directory belonging to the user, where Netscape will create its state files (.netscape and nsfiles).

We could easily have retained the real path to user’s home directory (e.g. /users/bob) so that the entry for the user in the /etc/pwd.db did not have to be changed. In that case, directory /bob would have to be changed to /users/bob. However, since we are replacing the /etc/pwd.db file anyway, we can change the user information in the copy and make the sandbox layout a bit simpler.

For more complex file access scenarios, it may be easier to be able to specify unprotected file hierarchies within the ACL file. This is still an open question which we plan to investigate as we gather more experience through the actual use of the FMAC system.

When using file substitutions it is often desirable to be able to use the replacement files in the passive phase as well. In our earlier example, where we changed Bob’s home directory, we wouldn’t really be able to get sensible data out of the FMAC tool, unless the changed /etc/pwd.db was used.

We solve this problem by allowing the user to specify a template ACL file when running the FMAC tool in passive mode. This template file would contain file substitutions like the ones mentioned above. The contents of the template file are included in the ACL produced by FMAC, so it is not necessary to include the template file when running FMAC in active mode.

6. Discussion

Access control, like other security related problems has no clear-cut solution; rather, it requires a compromise that balances costs against perceived risk.

The costs include execution overheads, memory requirements, application configuration, and, most importantly, the definition of the capabilities that the restricted process is allowed to have. For example, setting up a Windows 2000 system under VMware in order to read files generated by Microsoft Word requires an investment in time and expertise that may not be justified by the end result.

The FMAC tools attempt to strike an acceptable balance by providing:

Ease of configuration. Allowing the FMAC tool to learn from the trial runs of the application, reduces the initial work required for constructing the sandbox. Anybody who has manually configured an anonymous ftp server that uses the `chroot` facility can testify how difficult it is to determine the files and the permissions required for the its correct operation.

Portability. By being a user level program, FMAC has minimal installation overhead and is largely independent of the release or type of the operating system.

Security. The access restrictions imposed by the FMAC tool are in addition to the restrictions that are placed by the underlying file system. Thus, using the FMAC tool will not degrade the security posture of the system.

In the next paragraphs we will discuss some of the problems and caveats that are associated with the use of the FMAC tools.

The `chroot` and `mount` calls require superuser access. It is, therefore, imperative that we lower the capabilities of the process immediately after the two calls complete and before `execing` the application. We must also make sure that no hooks exist in the sandbox that may allow the boxed application to

escape. For example, although a shell must be present within the sandbox, we can use a restricted shell that provides only the bare essentials for the execution of the program. Users may not even have access to this program as they will be talking to the application. Moreover, files served via the FMAC system always have their `suid` and `sgid` bits cleared.

Particular care must be paid when running the application with the FMAC system in passive mode. During this phase, the application runs with all the privileges enjoyed by the user. Hostile activity on the part of the application will not be detected and may affect the security posture of the application when it runs with the FMAC in active mode.

Creating an access list file which is as complete as possible is also important because it reduces the need to go back and update the access list file later on. The user should try to use all the features of the application that are likely to be required in the future. For example, one feature that is seldom used during the active phase is the on-line help system. It is often required during the operational lifetime of the system, so it must be exercised during the passive phase so that the access list allows access to the help files.

The primary objective of the FMAC system is to prevent a user-level application such as Netscape Communicator from performing tasks that adversely affect the security of the user running the program, or the security of the machine hosting the application. Confining the program to a sandbox significantly reduces the possibility of undesired side effects. However, it is not a guarantee. A large and complicated program such as Netscape Communicator interacts with the system in many ways and it depends on a large number of system resources for its correct execution. For example when we follow a URL leading to the PDF file, the browser will automatically launch the Acrobat reader to display the page. Postscript files, streaming audio, video, etc. all require their own special helper applications. Placing inside the sandbox all the programs and devices that the helper programs need for their correct operation, will essentially negate the use of the sandbox.

On the other hand implementing workarounds for performing all these special tasks in a secure way, involves disproportionate amounts of work. For example, the simple act of sending a Web page to the printer involves the execution of `lpr` which is `setuid` root. The FMAC system will not allow `lpr` to run as root and the operation will fail. This may be overcome by depositing files in a "spool" directory

and having a daemon running outside the sandbox send them to the printer. Clearly this will appeal to few people and even if deemed adequate, implementing it will add to the overhead of configuring the sandbox.

Even if everything is configured properly, private information may still leak. In the Communicator example, email messages will need to be stored inside the sandbox so that the email application can access them. Downloaded files, cookies and bookmarks may also contain private information. All these can be accessed by malicious code that manages to subvert the application.

One workaround is to have automated scripts regularly move files out of the sandbox and clean up files that may contain private information.

Another category of malicious behavior that cannot be trapped is that which exploits the application capabilities in a manner that is roughly consistent with the intended use of the application. One example is the Melissa virus which sent copies of itself to email addresses contained in the user's address book.

Given the above, it is evident that FMAC is not a panacea. Rather it is yet another mechanism that can protect the user under certain circumstances. In particular, the ability of FMAC to rapidly create a disposable environment in which to run a potentially nasty applet, or contact a suspicious site, makes it an extremely useful tool.

7. Related work

Over the years there have been numerous proposals for systems that impose discretionary access controls on programs. These systems can be roughly placed in three categories, in increasing order of complexity for the execution environment.

Systems that trust programs. Programs that have been vetted are considered trusted, while the rest are given only limited access. Examples include Microsoft Active X controls and the system presented by Lai et al [3]. These systems assume that all bugs or vulnerabilities can be detected before deployment. This assumption has been demonstrated time and time again as utopian. In [3] only thirty-two programs from the entire BSD 4.3 were considered trusted. Ironically, one of them was `/etc/fingerd`, a daemon later used by the Morris Internet Worm to break into systems.

Regulate file access. File access is considered a key capability in most systems since it involves the long

term memory of the system. Unauthorized modifications to files can threaten the integrity of the system itself or the data that is stored in it. Even read-only access can be used to leak information to hostile parties. Controlling file accesses is, therefore, appealing both because it has significant impact and because file systems are often self contained OS subsystems that can be controlled with minimal modifications to the core operating system. Numerous systems are included in this category. The system described in [4] bases decisions on the file types (via the filename extension), while in the Exokernel [5], a credential-based system is used to specify file access. Credentials are used to determine which parts of the file hierarchy are accessible by an application. The system, however, is rather limited by the fact that permissions are hardwired into the system, the hierarchical capability tree may be up to eight levels deep, and the access-list based control mechanism is inflexible. Wichers et al [6] looked at the problem the other way round by attaching to files lists of programs that could access them.

FMAC relies on a custom filesystem providing the learning and chrooted access functionality. A number of projects have provided ways to create such filesystems, see [7] and the references therein.

Full access control. All requests made by the application are passed through a discretionary access control mechanism that enforces policy. The checks may be at the operating system call level as in Janus [1] and SubOS [8], or at the library call level [9]. Virtual machines such as the Java VM and VMware also provide a restricted environment in which programs may operate. Errant applications should only be able to cause damage to the virtual machine leaving the real system intact.

Recent versions of FreeBSD include the `jail` system call which is a more powerful version of the `chroot` facility that has been mentioned earlier. Like `chroot`, `jail` restricts the controlled process to a subset of the filesystem, but it also prevents the process and its children from issuing privileged requests such as creating device special nodes. The `jail` facility imposes a fixed access policy that cannot be altered without changing the implementation. Like a leash with a fixed collar its effective use is limited by the lack of flexibility.

Mobile code systems have to face many similar problems because they have to accommodate applications that are imported from the outside and hence are potentially hostile. The SANE architecture [10] includes a credential-based capability mechanism, while others [11, 12, 13] propose

Advisories from CERT and postings on security related forums provide ample evidence that many of the above systems fail to provide foolproof security. The guardians themselves often have flaws that may allow applications to escape from the sandbox and compromise system security.

8. Current Status and Future Directions

Both versions of the FMAC tools are currently fully operational. In addition to Netscape Communicator, the FMAC tools have been used to sandbox the Adobe Acrobat PDF reader and the ghostview application. We are currently investigating a mechanism whereby applications may negotiate an acceptable set of permissions with the FMAC system before executing, thus dispensing with the need to run the application in passive mode to construct the access list. Moreover, in the current system, files may not be added to the access list after the sandbox is running. We plan to investigate the possibility of asking the user for permission when trying to access a file that is not in the access list.

We intend to continue work on the system aiming at creating a fully fledged discretionary access control system for files. Moreover, we are currently investigating ways of controlling access to the network by dynamically creating special rules for the packet filtering facility in the kernel.

Acknowledgments

We would like to acknowledge the invaluable help of Sotiris Ioannidis and the other members of the DSL Lab at the University of Pennsylvania. Special thanks are also due to Ted Faber for his careful reading of the draft and his many right-to-the-point comments.

References

- [1] Goldberg, Ian, David Wagner, Randi Thomas and Eric A. Brewer, "A Secure Environment for Untrusted Helper Applications," 1996 USENIX Security Symposium.
- [2] Calvelli, Claudio, "The Perl Filesystem", <http://dd-sh.assurdo.com/perlfs>, April 2001.
- [3] Lai, N. and T.E. Gray, "Strengthening discretionary access controls to inhibit Trojan horses and computer viruses," Proceedings of the 1988 USENIX Summer Symposium, pages 275-286, June 1988.
- [4] Karger, P.A., "Limiting the damage potential of discretionary Trojan Horses," Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy," pages 32-337, April 1987.
- [5] David Mazieres and M. Frans Kasshoek, "Secure Applications Need Flexible Operating Systems," *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997
- [6] Wichers, D., D. Cook, R. Olsson, J. Cossley, P. Kerchen and R. Lo, "PACLSs: An Access Control List Approach to Anti-Viral Security," Proceedings of the USENIX SECURITY II Workshop, pages 71-82, 1990.
- [7] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. *USENIX Annual Technical Conference*, Anaheim, California, January 1997.
- [8] Ioannidis, Sotiris, Angelos D. Keromytis, Steve Bellovin and Jonathan M. Smith, "Implementing a Distributed Firewall," 7th ACM Conference on Computer Communications Security, November 2000.
- [9] Ko, Calvin, George Fink and Karl Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," Proceedings of the 10th Annual Computer Security Applications Conference, Orlando, FL, 1994
- [10] Alexander, D. Scott, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith, "A Secure Active Network Architecture: Realization in SwitchWare". IEEE Network Special Issue on Active and Controllable Networks, vol. 12 no. 3, pp. 37-45.
- [11] Edjlali, Guy, Anurag Acharya, and Vipin Chaudley, "History- Based Access for Mobile Code," In the Proceedings of the 5th ACM conference on Computer and Communication Security (CCS'98). 1998
- [12] Jaeger, T., A. Prakash, and A. Rubin, "Building Systems that Flexibly Control Downloaded Executable Context," Proceedings of the 6th USENIX Security Symposium, 1996.
- [13] Jajodia, S., P. Samarati, V. Subrahmanian, and E. Bertino, "A Unified Framework for Enforcing Multiple Access Control Policies," Proceedings of the ACM SIGMOD International Conference on the Management of Data, pages 474-485, 1997.

Building a Secure Web Browser*

Sotiris Ioannidis
sotiris@dsl.cis.upenn.edu
University of Pennsylvania

Steven M. Bellovin
smb@research.att.com
AT&T Labs Research

Abstract

Over the last several years, popular applications such as Microsoft Internet Explorer and Netscape Navigator have become prime targets of attacks. These applications are targeted because their function is to process unauthenticated network data that often carry active content. The processing is done either by helper applications, or by the web browser itself. In both cases the software is often too complex to be bug free. To make matters worse, the underlying operating system can do very little to protect the users against such attacks since the software is running with the user's privileges.

We present the architecture of a secure browser, designed to handle attacks by incoming malicious objects. Our design is based on an operating system that offers process-specific protection mechanisms.

Keywords: Secure systems, web browser, process-specific protection.

1 Introduction

In the current highly interconnected computing environments, Web browsers are probably the most popular tool for receiving data over the internet. More often than not, the data come from unauthenticated sources that can potentially be malicious. Since the incoming data often carry active content that will be interpreted on the client machine, in many cases without the users knowledge, a number of attacks become possible.

To interpret active content web browsers often rely on helper applications, that become security critical

since they operate on untrusted data. These applications which are often buggy [11], execute with the users privileges and can therefore compromise the security of the system. Furthermore the browsers also interpret code like JavaScript and VBScript [6], making the browser itself vulnerable ¹.

In this paper we present the architecture of a secure web browser. Our system is designed to address the problems that plague the popular Web browsers by using support offered by the operating system. We built our prototype on SubOS [12]. SubOS is an operating system that offers process-specific protection mechanisms, which we will explain in Section 3.

The paper is organized as follows. In Section 2 we discuss the motivation behind this work. In Section 3 we give a brief background description of a SubOS-capable operating system. In Section 4 we present the architecture of our system. In Section 5 we discuss related work, and finally we conclude in Section 6.

2 Motivation

With the growth of the Internet, exchange of information over wide-area networks has become essential for users. Web browsers, like Netscape Navigator and Microsoft Internet explorer often automatically invoke helper application to handle the downloaded object. In some cases, like in Perl scripts, they will query the user before executing it. In others, like in Postscript files or Java applets [10, 15, 9], they will execute the content, possibly compromising the security of the system. The former approach puts a lot of burden on the user, who more often than not is not particularly security conscious. In

*This work was supported by DARPA under Contract F39502-99-1-0512-MOD P0001.

¹There are a number of hostile JavaScript and VBScript sites on the Web, easily found using search engines

the latter case the user is bypassed altogether and system security becomes dependent on the correctness of the Postscript or Applet viewer.

It is also the case that seemingly inactive objects like Web pages are very much active and potentially dangerous. One example is JavaScript [6] programs which are executed within the security context of the page with which they were down-loaded, and they have restricted access to other resources within the browser. Security flaws exist in certain Web browsers that permit JavaScript programs to monitor a user's browser activities beyond the security context of the page with which the program was down-loaded (CERT Advisory CA:97.20). It is obvious that such behavior automatically compromises the user's privacy and security.

The lack of flexibility in modern operating systems is one of the main reasons security is compromised. The UNIX operating system, in particular, violates the principle of least privilege. The principle of least privilege states that a process should have access to the smallest number of objects necessary to accomplish a given task. UNIX only supports two privilege levels: "root" and "any user".

To overcome this shortcoming, UNIX, can grant temporary privileges, namely `setuid(2)` (set user id) and `setgid(2)` (set group id). These commands allow a program's user to gain the access rights of the program's owner. However, special care must be taken any time these primitives are used, and as experience has shown a lack of sufficient caution is often exploited [13].

Another technique used by UNIX is to change the apparent root of the file system using `chroot(2)`. This causes the root of a file system hierarchy visible to a process to be replaced by a subdirectory. One such application is the `ftpd(8)` daemon; it has full rights in a safe subdirectory, but it cannot access anything beyond that. This approach, however, is very limiting, and in the particular example commands such as `ls(1)` become unreachable and have to be replicated.

These mechanisms are inadequate to handle the complex security needs of today's applications. This forces a lot of access control and validity decisions to user-level software that runs with the full privileges of the invoking user. To overcome these shortcomings applications such as Web browsers become responsible for accepting requests, granting permis-

sions and managing resources. All this is what is traditionally done by operating systems. Web browsers consequently, because of their complexity as well as the lack of flexibility in the underlying security mechanisms, possess a number of security holes. Examples of such problems are numerous, e.g. JavaScript, malicious Postscript documents, etc.

We wish to demonstrate how to build a secure browser, designed to handle attacks by incoming malicious objects, on top of an operating system that offers process-specific protection mechanisms.

3 SubOS-enabled Operating Systems

SubOS is a process-specific protection mechanism, a more extensive discussion on SubOS can be found in [12]. Under SubOS any application (e.g. ghostscript, perl, etc.) that might operate on possibly malicious objects (e.g. postscript files, perl scripts, etc.) behaves like an operating system, restricting their accesses to system resources. We are going to call these applications SubOS processes, or sub-processes in the rest of this paper. Figures 1 and 2 demonstrate the difference between a regular and a SubOS-enabled operating system. The access rights for that object are determined by a sub-user id that is assigned to it when it is first accepted by the system. The sub-user id is a similar notion to the regular UNIX user id's. In UNIX the user id determines what resources the *user* is allowed to have access to, in SubOS the sub-user id determines what resources the *object* is allowed to have access to. The advantage of using sub-user id's is that we can identify individual objects with an immutable tag, which allows us to bind a set of access rights to them. This allows for finer grain per-object access control, as opposed to per-user access control.

The idea becomes clear if we look at the example shown in Figure 3. Let us assume that our untrusted object is a postscript file `foo.ps`. To that object we have associated a sub-user id, as we will discuss in Section 3.1. `foo.ps` initially is an inactive object in the file system. While it remains inactive it poses no threat to the security of the system. However the moment `gs(1)` opens it, and starts executing its code, `foo.ps` becomes active, and automatically a possible danger to the system. To contain this threat, the applications that open untrusted ob-

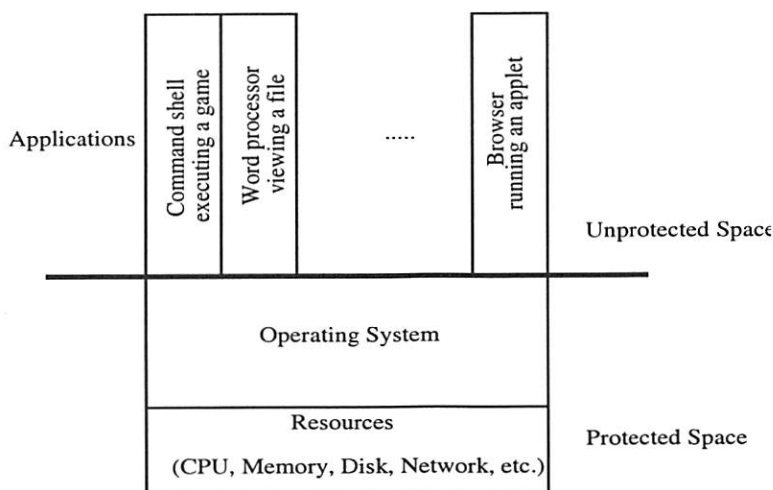


Figure 1: User applications executing on an operating system maintain the user privileges, allowing them almost full access to the underlying operating system.

jects, inherit the sub-user id of that objects, and are hereafter bound to the permissions and privileges dictated by that sub-user id.

There is a strong analogy here to the standard UNIX `setuid(2)` mechanism. When a suitably-marked file is executed, the process acquires the access rights of the owner. With SubOS, suitably-marked *processes* acquire the access rights of the owner of the *files* that they open. In this case, of course, the new rights are never greater than those the process had before.

The advantages of our approach become apparent if we consider the alternative methods of ensuring that a malicious object does not harm the system. Again using our postscript example we can execute `foo.ps` inside a safe interpreter that will limit its access to the underlying file system. There are however a number of examples on how relying on safe languages fails [11]. We could execute the postscript interpreter inside a sandbox using `chroot(2)`, but this will prohibit it from accessing font files that it might need. Finally we could read the postscript code and make sure that it does not include any malicious commands, but this is impractical.

Our method provides transparency to the user and increased security since every data object has its access rights bound to its identity, preventing it from harming the system.

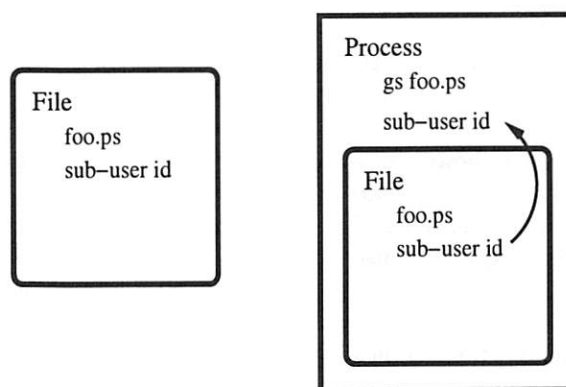


Figure 3: In the left part of the Figure we see an object, in this case a postscript file `foo.ps`, with its associated sub-user id. The moment the ghostscript application opens file `foo.ps`, it turns into a SubOS process and it inherits the sub-user id that was associated with the untrusted object. From now on, this process has the permissions and privileges associated with this sub-user id.

3.1 Security Mechanism Enforcement

As we mentioned earlier in Section 3, every time the system accepts an incoming object it associates a sub-user id with it, depending on the credentials the object carries. The sub-user id is permanently saved in the Inode of the file that holds that object, which is now its immutable identity in the system and specifies what permissions it will have. It has essentially the same functionality as a UNIX user

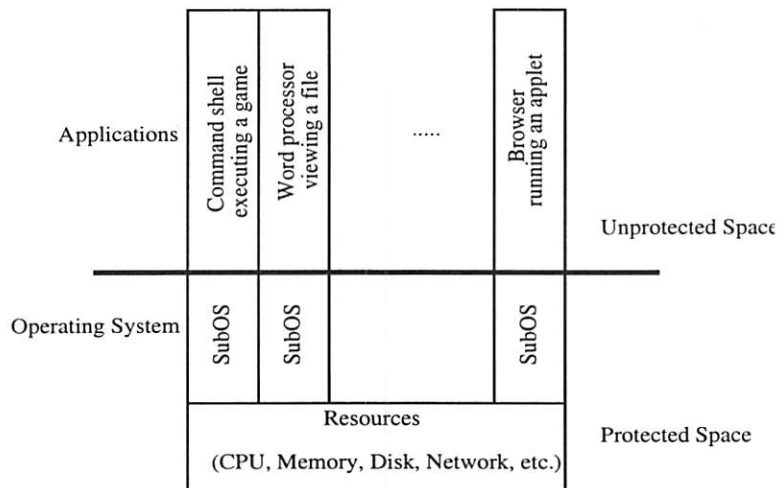


Figure 2: Under SubOS enabled operating systems user applications that “touch” possibly malicious objects no longer maintain the user access rights, and only get restricted access to the underlying system.

id. One can view this as the equivalent of a user logging in to the system.

Figure 4 shows the equivalence of the two mechanisms. In the top part of the figure we see the regular process of a user Bar logging in a UNIX system Foo and getting a user id. In the same way, objects that enter the system through ftp, mail, etc., “log in” and are assigned sub-user id’s based on their (often cryptographically-verified) source.

4 The Browser Architecture

4.1 The Threat

The use of Java, JavaScript and VBScript in HTML pages is becoming ever more popular, furthermore HTML provides support for other scripting languages with the use of the <SCRIPT> tag [3]. Even though this functionality is primarily intended to enhance the capabilities of web pages and the “surfing experience” of the user, it is often used to attack unsuspecting hosts.

Even worse, the site or host is vulnerable even if the browser is behind the firewall and the document is a “secure” HTTPS-based document. JavaScript programs are executed within the security context of the page in which they were down-loaded,

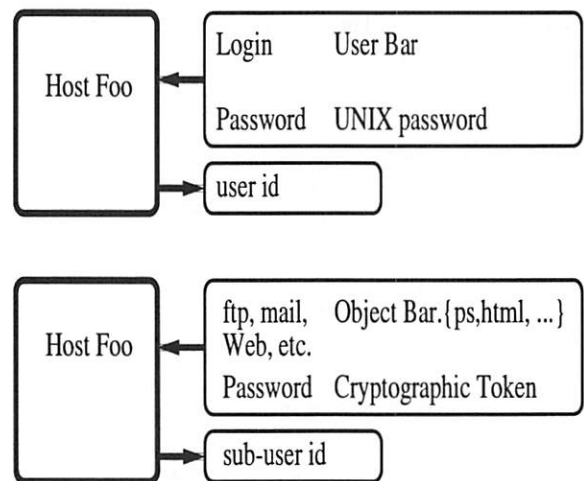


Figure 4: In the top part of the Figure we see the regular process of a user Bar logging in a UNIX system Foo and getting a user id. In the same way objects that enter the system through ftp, mail, etc., “log in” using a cryptographic token, and are assigned sub-user id’s.

and should have restricted access to other resources within the browser. Some browsers running JavaScript may, in turn, have security flaws that allow the JavaScript program to monitor a user’s browser more than what is considered safe or secure. In addition, it may be difficult or impossible for the browser user to determine if the program is transmitting information back to the web server.

For instance, among other functions, JavaScript is able to monitor a user's browser activity by:

- Observing the URLs of visited documents as well as bookmarks.
- Observing the data filled into HTML forms (including passwords).
- Observing the values of cookies (that might hold critical information).

In Java the user may or may not be informed that an applet is being down-loaded into their browser. The real shock comes when a user inadvertently down-loads a hostile applet. There are many different things hostile applets can do to wreak havoc on your system. Among a few of the most noteworthy are the following:

- Reveal information about your machine (e.g. details about passwords or structure of your system).
- Allocate resources to the point your machine "locks up" (i.e. denial of service attacks).
- Delete or alter files.
- Be just plain annoying (e.g. popping countless windows).

Hostile applets have also been known to have the capability to contact machines behind firewalls, send off a listing of a user's directories, track a user's actions through the web, generate machine code, make directories readable and writable, and send off email without intention ².

4.2 Modular Approach

In our architecture we address the two security problems of Web browsers:

1. Helper applications running with the user's privileges.

²There are a number of web sites that list hostile applets, JavaScript and VBScript, readily available for anyone interested in launching an attack.

2. Web pages that carry active content that is interpreted by the browser.

To address these problems we will use the mechanisms provided by the SubOS-capable operating system, as well as a modular Web browser architecture. We divide the Web browser into three parts, according to its functionality. The first part is responsible for down-loading objects over the network, the second is responsible for displaying the content, and the last is a set of helper applications/interpreters used to process the content of the down-loaded objects. The design is presented in Figure 5

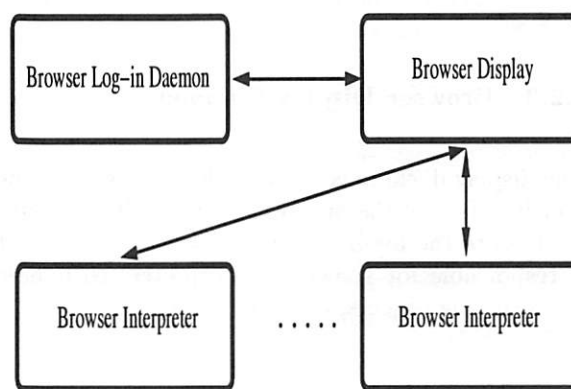


Figure 5: The Web browser is comprised of three parts. The first part is responsible for down-loading objects from the net and assigning sub-user id's to them. The second provides the user interface of the browser. Finally the third is a set of processes that interpret the active code that is carried by the incoming objects.

We decided against using an existing Web browser since that would require significant modification to its architecture. Down-loading and authentication of objects could be easily achieved by using a proxy, however execution of embedded code in HTML web pages would be a lot more challenging, since it would have to execute in a separate address space to maintain its security properties, as we discussed in Section 3.

4.2.1 Browser Log-in Daemon

Every object that is down-loaded by our browser log-in daemon is assigned a sub-user id, which is bound to some permissions, and is then stored in the

file system. Assignment of sub-user id's is similar to the log in mechanism of UNIX. Objects that carry certificates are given more permissions than unauthenticated objects. For example an authenticated object might get access to `/home/user_foobar`, network access and unlimited resources, whereas an unauthenticated objects might only get access to `/tmp` with no access to the network and limited CPU time and memory allocation.

In the current implementation we use the URL address is used to select the sub-user id that will be assigned to the down-loaded object. This approach of course is not really secure, ideally we should use some sort of cryptographic token (e.g. a certificate) that is carried along with the down-loaded object.

4.2.2 Browser Display Daemon

The display daemon is responsible for providing the user interface of the our Web browser. It can make requests to the log-in daemon to down-load files, it is responsible for spawning interpreters to handle the incoming objects, and display HTML.

4.2.3 Browser Interpreter Daemon

The final part of our web browser is the set of interpreter daemons. These processes have dual functionality; they interpret HTML along with any possible active content embedded in the web page, and they execute the helper applications that handle incoming objects such as Perl, Postscript, etc.

Objects that are normally handled by helper applications are also assigned sub-user id's by the log-in daemon, the same way as ordinary web pages. When they are interpreted they are bound to the permissions of that sub-user id. This way users don't need to be queried about every arbitrary object they down-load of the net and also don't have to worry about executing possibly malicious code on their machine.

When the interpreter daemon encounters active code embedded in a web page (by encountering an `<APPLET>` or `<SCRIPT>` tag) it spawns a process to interpret the Java, JavaScript [1], or Perl code. The new process inherits the permissions of the parent process so the active code can never escape it's sandbox.

5 Related Work

Web browser security is topic that has received a great deal of attention since its so crucial in today's highly interconnected computing. However there have not been any satisfactory solutions so far. The primary proposed solution is secure interpreters for JavaScript, VBScript, Java, etc. [14, 15, 17, 10, 9]. Such solutions fail because of their complexity. The more complex the implementation, the more likely it is to have bugs. Furthermore they don't address the issue of other helper applications like Perl or Tcl. When they are invoked, the user is asked to give a blanket "go" "no-go" response, and this puts a lot of burden to the user.

Another language related technique used for ensuring security is code verification. This approach uses *proof-carrying code* [16] to demonstrate the security properties of the object. This means that the object needs to carry with it a formal proof of its properties; this proof can be used by the system that accepts it to ensure that it is not malicious. Code verification is very limiting since it is hard to create such proofs. Furthermore, it does not scale well; imagine creating a formal proof for every Web page.

A different approach relies on the notion of system call interception, as used by systems such as TRON [5], MAPbox [4], Software Wrappers [7] and Janus [8]. TRON and Software Wrappers enforce capabilities by using system call wrappers compiled into the operating system kernel. The syscall table is modified to route control to the appropriate TRON wrapper for each system call. The wrappers are responsible for ensuring that the process that invoked the system call has the necessary permissions. The Janus and MAPbox systems implement a user-level system call interception mechanism. It is aimed at confining helper applications (such as those launched by Web browsers) so that they are restricted in their use of system calls. To accomplish this they use `ptrace(2)` and the `/proc` file system, which allows their tracer to register a callback that is executed whenever the tracee issues a system call. These systems are the most related to our work; however, our system differs in a major point. We view every object as a separate user, each with its own sub-user id and access rights to the system resources. This sub-user id is attached to every incoming object when it is accepted by the system, and stays with it throughout it's life, making it impossible for malicious objects to escape.

6 Conclusions

We have presented the architecture of a secure web browser, that protects against malicious incoming objects. We have implemented a first version of our prototype on a SubOS-capable OpenBSD 2.8 [2] operating system using Perl.

There are several advantages in our modular architecture versus the monolithic architecture of popular Web browsers, such as Netscape Navigator and Microsoft Internet Explorer. Our design adds a stage of authentication before any incoming object is processed. The burden of access control is moved from the browser and its helper applications, to the operating system, allowing for a simpler and therefore more secure design. Finally the user is not involved in the processing of incoming objects, and therefore cannot be tricked into executing hostile code. Presently however, our architecture requires that the operating system provides a data centric protection mechanism, that associates permissions and privileges to data objects. This limits us to our experimental SubOS-enabled OpenBSD operating system.

There are still some things that remain to be added to our prototype browser in order to offer more complete functionality:

- We currently don't support frames. Frames require special handling since each frame consists of an HTML document with possibly individual security properties. In future versions of our browser we will add this functionality to the browser display daemon.
- Only a subset of HTML was implemented so there are a number of tags that need to be added, along with their possible variables.
- We want to expand the <SCRIPT> tag to deal with additional embedded scripting languages other than JavaScript and Perl.
- Finally we need to have some kind of secure authentication mechanism for the browser log-in daemon. The possible solutions we are considering are either an additional tag that carries a certificate in the down-loaded web page, or a certificate attached to the HTTP request.

7 Acknowledgments

We would like to thank Jonathan M. Smith for his useful comments and guidance throughout the course of this work. We also like to thank the FREENIX 2001 anonymous reviewers and our "shepherd" Ken Coar for their comments and suggestions on improving this paper.

References

- [1] NJS JavaScript Interpreter. <http://www.bbassett.net/njs/>.
- [2] The OpenBSD Operating System. <http://www.openbsd.org/>.
- [3] World Wide Web Consortium. <http://www.w3.org/>.
- [4] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 2000 USENIX Security Symposium*, pages 1–17, Denver, CO, August 2000.
- [5] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *USENIX 1995 Technical Conference*, New Orleans, Louisiana, January 1995.
- [6] David Flanagan. *JavaScript The Definitive Guide*. O'Reilly, 1998.
- [7] Tim Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [8] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX 1996 Technical Conference*, 1996.
- [9] Li Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
- [10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.
- [11] <http://www.cert.org/advisories/>.

- [12] Sotiris Ioannidis and Steven M. Bellovin. Sub-Operating Systems: A New Approach to Application Security. Technical Report MS-CIS-01-06, University of Pennsylvania, February 2000.
- [13] R. Kaplan. SUID and SGID Based Attacks on UNIX: a Look at One Form of their Use and Abuse of Privileges. *Computer Security Journal*, 9(1):73–7, 1993.
- [14] Jacob Y. Levy, Laurent Demailly, John K. Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. In *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.
- [15] Gary McGraw and Edward W. Felten. *Java Security: hostile applets, holes and antidotes*. Wiley, New York, NY, 1997.
- [16] G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Lecture Notes in Computer Science Special Issue on Mobile Agents*, October 1997.
- [17] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

Citrus project: true multilingual support for BSD operating systems

Jun-ichiro itojun Hagino <itojun@ijlab.net>

Research Laboratory, Internet Initiative Japan Inc.
<http://citrus.bsdclub.org/index.html>

Abstract

Citrus project aims to implement a complete multilingual programming environment for BSD-based operating systems. The goals include:

- ISO C/SUS V2-compatible multilingual programming environment (locale support),
- multi-script framework, which decouples C API and actual external/internal encoding,
- gettext and POSIX NLS catalog,

All of our source code is, and will be distributed under a BSD-like license.

The paper concentrates onto the multi-script framework, which is unique and central to our approach. Most other free software implementations support only Unicode in their multilingual library, or converts external representation into Unicode internal representation and loses significant information on the external representation. We believe a Unicode-only approach is not future-proven, and is not the right way to handle multilingual text. We support multiple different encodings in ISO C/SUS V2 compatible library, and made our library code (as well as user programs) future-proven.

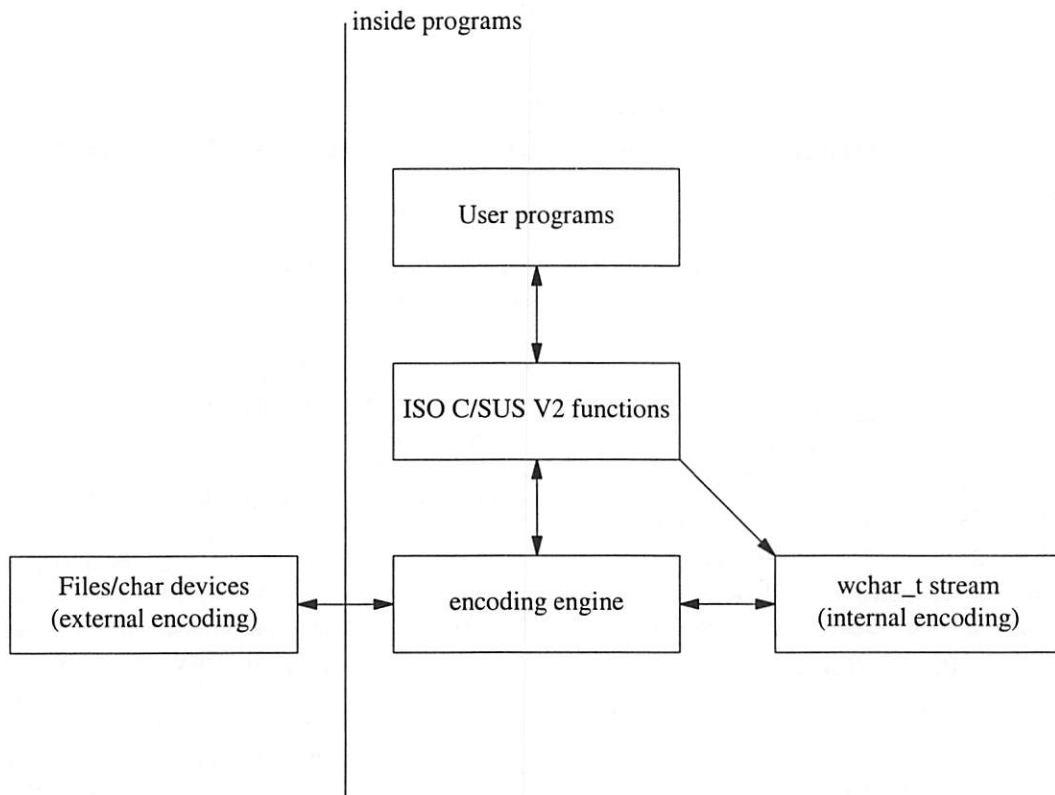
1. Motivation, and multi-script framework

Compared to vendor UN*X operating systems and GNU libc, BSD-based operating systems has been a bit behind in multilingual programming support. This may be because of lack of manpower, difference in interest, or whatever. Anyway, because of the lack of multilingual support, we are seeing increasing pain in porting applications from/to BSD-based operating systems. For example, GNOME, GTK+ and other window manager software relies heavily upon the presence of multilingual libraries. We definitely need a multilingual support library for BSD-based operating systems that is based on ISO C/SUS V2 standards.

The ISO C/SUS V2 standard uses two different encodings for multilingual support: an internal and an external encoding. We use the term "internal encoding" to mean a multilingual encoding system used inside C programs (like inside variables for manipulation), normally declared as an array of `wchar_t`. On other papers the term "process code" is also used. "External encoding" means a multilingual encoding system used outside of programs (like files or screen output stream). It is also called as "file code". The ISO C/SUS V2 libraries will supply conversion functions between external and internal encoding. Here we call the functions "encoding engine".

Many of existing ISO C/SUS V2 libraries, especially freely available ones like GNU libc, assume that the internal encoding is Unicode-based. More specifically, they assume UCS4 as the internal encoding. They also advocate UTF-8 [Yergeau, 1998] as the external encoding. The encoding engine is hardcoded for internal UCS4 encoding. External encodings other than UTF-8 are supported by conversion functions in encoding engine, which converts external encodings to UCS4, and vice versa. To implement a correct multilingual support, we believe it is very important to not hardcode any encoding, including Unicode. The Citrus project library does not hardcode any existing encoding. In the next section we discuss more fully why Unicode is not enough.

In order to be able to make less assumptions about multilingual encodings, we have designed our libraries to support multiple external encodings, and multiple encoding engines and internal encodings. In other words, each internal encoding has its own encoding engine. We call this concept a "multi-script framework." To support multiple encodings, we simply need to supply the appropriate encoding engines. We also use dynamic loading to load encoding engines, so that we can add more engines on the fly.



2. Why Unicode is not enough

You may be still wondering why we need to support a multi-script framework, instead of hard-coded Unicode support. Below we discuss why Unicode is not sufficient, and why hardcoding of encodings is not preferable.

We specifically have chosen NOT to hard-code Unicode in our library suite. Since we started using computers for text processing, we have experienced many transitions in character set encodings. Here we list our history in chronological order:

- First, we had a total chaos of vendor-defined character set encodings, with 6bit, 7bit or 8bit encodings.
- ASCII-only 7bit encodings (and EBCDIC in some places).
- Hardcoded 8bit encodings. For example, in many of the european countries, ASCII + ISO-8859-1 (so-called Latin-1) has been used. In Japan, JIS X0201, which gives us ASCII variant and Katakana, was widely used.
- Stateful ISO2022 character encodings, with explicit character set designations.

ISO-2022 character encodings allow us to encode multiple language text into a single

plaintext stream, and is a good foundation for truly internationalized plaintext handling. For example, by using X11 ctext encoding (which is a subset of ISO-2022 encoding), we can mix Korean, Chinese, Taiwanese, and Japanese text into a single plaintext stream.

Separately from the above direction, there were a couple of regional encodings (encodings that support single language only within a single plaintext stream), including EUC (like euc-kr), MS-Kanji (Shift JIS), or Unicode. Here we list Unicode under "regional encodings", as we cannot mix Chinese/Taiwanese/Japanese text in a plaintext - we need to annotate plaintext with font designation to do it.

Unicode cannot handle multiple Asian characters in a plaintext at the same time, due to "han unification". Unicode maps multiple characters from different Asian regions, into the same Unicode codepoint - this is called "han unification". It was introduced to reduce the amount of codepoint used in Unicode (to fit characters into UCS2 16bit region). While some of the unified characters are indeed same across Asian regions, others have totally different glyphs and meanings in different Asian regions [KUBOTA,]. Suppose that we have assigned the same codepoint for O, and O with umlaut. Some people cannot notice

the difference, however, this will become a significant problem for people who use O with umlaut in their language (like for German language). Also note that, if we convert Asian multilingual text into Unicode and then convert it back to other multilingual encoding, the conversion will not be able to preserve the information contained in the original multilingual text, due to the han unification. When we convert the multilingual text into Unicode, we will lose the information encoded in the source due to the han unification.

NOTE: there are proposals to perform language tagging [Whistler, 1999] in Unicode, however, the language tagging jeopardizes one of the very important aspects of Unicode, uniform 32bit wide-char representation, and we do not consider it to be useful.

Every time we change from one encoding system to another, the transition is very painful. In fact, there still are applications that are not 8bit-clean. Even for Unicode, there are implementations assume 16bit UCS2, and they need to migrate to 32bit UCS4. From our experience, it makes no sense any more to pick a single encoding to rely on. We do not advocate the use of ISO-2022, either. We believe that no encodings should be hardcoded, since to do so means that we will have to bear painful transitions over and over again.

Here is another reason for us to avoid hardcoded encoding, and avoid hardcoded Unicode support. There has been widely deployed user-base which uses non-Unicode multilingual text, including big5, euc-kr, KOI8-R, MS-Kanji and some other encodings. If someone says "all people just need to transition to Unicode", that is wrong. Unicode imposes no pain to Latin-1 user-base, while imposing huge pain to Asian and other non-Latin-1 people. And, even if we transition to Unicode, we are unsure if it is going to be enough. So we conclude that we should hardcode no encodings.

What we should do is very similar to the approach with MIME [Freed, 1996]. We will have a way to identify encoding in a plaintext stream, and have support for multiple different encodings. We will switch encoding engines according to the encoding identification. In C library case, we identify encoding by locale settings made by `setlocale(3)`.

3. Gory details

Under our implementation based on multi-script framework, we can switch external encoding, internal encoding and encoding engine as we wish, and we support multiple encodings/engines.

External encoding is normally a stateful, or stateless multibyte encoding, like ISO-2022-JP [Murai, 1993] or UTF-8. An octet stream will be used for multilingual representation inside files. Many of the existing external encodings use variable-length encodings; one letter will be presented as a octet, two octets, or more octets. When we read in external encoding representation into C program, we normally use array of `char` to hold it. Internal encoding is a stateless, fixed-bitwidth encoding. It is defined internally by the library, depending on the current encoding engine we are using. We use a type called `wchar_t` to hold it. At this moment `wchar_t` is a 32bit integer (`int32_t`).

When the external encoding is 8bit (like Latin-1), we can just typecast external encoding (`char`) into internal encoding (`wchar_t`). When the external encoding is UTF-8, the natural choice for internal encoding is UCS4. RFC2279 [Yergeau, 1998] defines standard conversion between them. When the external encoding is ISO-2022, we use a compressed representation of ISO-2022 stream as the internal encoding.

With `setlocale(3)`, we pick a pair of internal and external encoding, and encoding engine. A programmer can convert internal encoding and external encoding, using ISO C/SUS V2-compatible library calls, like `mbstowcs(3)`.

User programs should manipulate `wchar_t` stream only, and should not manipulate text encoded with external encoding. The details of internal and external encoding are embedded into the library API. Programs should not, and need not to care about internal nor external encoding at all. If a programmer hardcodes some assumption about internal/external encoding to their programs, the programs will not be future-proven. We will also supply `wchar_t`-ready curses, regex and other libraries, to keep external encoding outside of user programs.

From our strategy, we have two major benefits. First, our library is future-proven, as long as internal encoding fits into the bitwidth of `wchar_t` (currently 32bit). Next benefit is that we can simplify encoding engine as we wish. Suppose we need to support JIS X0201 or Latin 1 as internal/external encodings. In this case, the encoding

engine can be a simple memory copy logic. We do not need to visit a slow encoding engine for simple encodings.

Our strategy avoids problem with Unicode and han unification. Since we can use specific encoding engine that matches the external and internal encodings, we can preserve information supplied by the external data representation, into internal data representation. Therefore, we can have no data lossage during conversion from external to internal encoding, or vice versa.

If we use Unicode as internal and external encoding, we would not be able to enjoy the above mentioned benefits. With UCS4 as the internal encoding, it is very hard to support external encodings other than UTF8. To support them, we would need a huge conversion table to convert the external encoding into the internal encoding, and vice versa. Also, it will not be possible to simplify the encoding engine, even if internal/external encodings are simple enough.

4. Multiple encodings

ISO C/SUS V2 multilingual programming API assumes that a single program uses single encoding throughout the program. External and internal encodings are picked when `setlocale(3)` is called, and the function usually gets called on program startup time. Normally, we cannot determine which encoding should be used, from a `wchar_t` data stream.

There are various applications that would need a library support for multiple encodings during their runtime session. Examples would be web clients, text editors and email readers. For these applications, we would need to pick internal and external encodings based on input data stream. So, ISO C/SUS V2 API is not sufficient for these type of applications.

We have a temporary workaround to provide a better multiple encodings support. ISO C/SUS V2 API has a data type, `mbstate_t`, to hold the intermediate state for encoding engine. Our implementation includes a hidden reference to encoding engine inside `mbstate_t`. By holding an `mbstate_t` variable with a `wchar_t` array, we can identify the encoding engine used to encode the `wchar_t` stream. When we convert `wchar_t` (internal encoding) back to external encoding, we can automatically use the appropriate encoding engine.

We still are investigating a better API to abstract the manipulation of multiple encodings in a single program. We would like to make a proposal when we are done.

5. Status of implementation

The Citrus project implementation is based on 4.4BSD `runelocale` implementation [Borman,]. The project expanded it significantly to support multi-script framework, `iconv(3)`, BSD-licensed `gettext` library, more locale supports like `LC_TIME`, and expansions for simultaneous multiple locale handling (as presented above). We still are missing multilingual support for stdio routines, like `fgetwc` function, as it require us to modify `FILE` structure on each of the BSD system. Depending on the details in standard I/O function implementation, the change to `FILE` structure may need a `libc` shared library major number bump. The current implementation supports NetBSD, OpenBSD and FreeBSD. We are trying to finalize our implementation and integrate it into BSDs. NetBSD integration is ahead of other BSDs at this moment. `LC_CTYPE` locale support and BSD-licensed `gettext` library were integrated into NetBSD development tree and will be available in NetBSD 1.6. The source code is available via anonymous CVS. The project is definitely an ongoing effort.

At this moment, we are using a tool called `mklocale(1)` for converting `LC_CTYPE` locale definition files into binary representation. The `mklocale(1)` tool and the locale definition file format are derived from `runelocale` implementation. We should migrate to more standard tool like `localedef(1)`, and the standard file format for locale definition files.

There still are couple of issues to be resolved. We picked 32bit `wchar_t` for now, just like many of other UNIX operating systems do. It is good enough for future? It is a good question. We believe 32bit is a good compromise. A good thing is that we actually are future-proven. As long as people do not hardcode the current assumption that `wchar_t` is of 32bit quantity, we will be able to expand `widechar` representation to 64bit, or something larger. It requires full recompilation of operating system and userland programs, but the transition will impose no change in code. The transition will be just like transitioning from 32bit `time_t` to 64bit.

For full locale support (including `LC_TIME` and `LC_COLLATE`), we will need a

large database for localized character tables, time format and others. For voluntary free software effort it can be way too hard to maintain. The maintenance cost for the LC_CTYPE locale databases is also high. We are wondering if we can integrate database from ICU [IBM,], and reuse it in our multi-script framework.

6. Conclusion

We at Citrus project aim to implement a complete multilingual programming environment for BSD-based operating system. The most important aspect of the effort is multi-script framework. We try to avoid any hardcoded encoding in our library, and embed conversions and encodings into ISO C/SUS V2 API.

DEC/Compaq Tru64Unix uses a similar technology as we have used [Compaq,], including multiple switchable internal encoding, dynamic library support for additional encodings support, and the use of 32bit wchar_t. As mentioned in the abstract, vendor UNIX implementations are very ahead of free software implementations, regarding to multilingual support. We wish to see more of vendor technologies to be made available to public consumption, under BSD or GNU license.

The author would like to thank Freenix reviewers including Wendy Rannenberg, and Citrus project members including Henry Nelson, for the time they gave me improve the paper.

References

- Yergeau, 1998.
F. Yergeau, "UTF-8, a transformation format of ISO 10646" in *RFC2279* (January 1998). <ftp://ftp.isi.edu/in-notes/rfc2279.txt>.
- KUBOTA, .
Tomohiro KUBOTA, *Most Important 1006 Ideographs for Japanese*.
<http://www.debian.or.jp/~kubota/unicode/>.
- Whistler, 1999.
K. Whistler and G. Adams, "Language Tagging in Unicode Plain Text" in *RFC2482* (January 1999). <ftp://ftp.isi.edu/in-notes/rfc2482.txt>.
- Freed, 1996.
N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies" in *RFC2045* (November 1996.).
<ftp://ftp.isi.edu/in-notes/rfc2045.txt>.
- Murai, 1993.
J. Murai, M. Crispin, and E. van der Poel, "Japanese Character Encoding for Internet Messages" in *RFC1468* (June 1993).
<ftp://ftp.isi.edu/in-notes/rfc1468.txt>.
- Borman, .
Paul Borman, *4.4BSD rune(3) implementation*.
- IBM, .
IBM, *ICU: International Components for Unicode*. <http://oss.software.ibm.com/developerworks/open-source/icu/project/>.
- Compaq, .
Compaq, "Writing software for the International Market" in *Tru64 UNIX Version 5.1 programming online documentation*.
http://tru64unix.compaq.com/faqs/publications/base_doc/DOCUMENTATION/V51_HTML/ARH9YBTE/TITLE.HTM.

Author's address

Jun-ichiro itojun HAGINO
Research Laboratory, Internet Initiative Japan Inc.
Takebashi Yasuda Bldg.,
3-13 Kanda Nishiki-cho,
Chiyoda-ku, Tokyo 101-0054, JAPAN
Tel: +81-3-5259-6350
Fax: +81-3-5259-6351
Email: itojun@iijlab.net

Kqueue: A generic and scalable event notification facility

Jonathan Lemon jlemon@FreeBSD.org
FreeBSD Project

Abstract

Applications running on a UNIX platform need to be notified when some activity occurs on a socket or other descriptor, and this is traditionally done with the `select()` or `poll()` system calls. However, it has been shown that the performance of these calls does not scale well with an increasing number of descriptors. These interfaces are also limited in the respect that they are unable to handle other potentially interesting activities that an application might be interested in, these might include signals, file system changes, and AIO completions. This paper presents a generic event delivery mechanism, which allows an application to select from a wide range of event sources, and be notified of activity on these sources in a scalable and efficient manner. The mechanism may be extended to cover future event sources without changing the application interface.

1 Introduction

Applications are often event driven, in that they perform their work in response to events or activity external to the application and which are subsequently delivered in some fashion. Thus the performance of an application often comes to depend on how efficiently it is able to detect and respond to these events.

FreeBSD provides two system calls for detecting activity on file descriptors, these are `poll()` and `select()`. However, neither of these calls scale very well as the number of descriptors being monitored for events becomes large. A high volume server that intends to handle several thousand descriptors quickly finds these calls becoming a bottleneck, leading to poor performance [1] [2] [10].

The set of events that the application may be interested in is not limited to activity on an open file descriptor. An application may also want to know when an asynchronous I/O (aio) request completes, when a signal is

delivered to the application, when a file in the filesystem changes in some fashion, or when a process exits. None of these are handled efficiently at the moment; signal delivery is limited and expensive, and the other events listed require an inefficient polling model. In addition, neither `poll()` nor `select()` can be used to collect these events, leading to increased code complexity due to use of multiple notification interfaces.

This paper presents a new mechanism that allows the application to register its interest in a specific event, and then efficiently collect the notification of the event at a later time. The set of events that this mechanism covers is shown to include not only those described above, but may also be extended to unforeseen event sources with no modification to the API.

The rest of this paper is structured as follows: Section 2 examines where the central bottleneck of `poll()` and `select()` is, Section 3 explains the design goals, and Section 4 presents the API of new mechanism. Section 5 details how to use the new API and provides some programming examples, while the kernel implementation is discussed in Section 6. Performance measurements for some applications are found in Section 7. Section 8 discusses related work, and the paper concludes with a summary in Section 9.

2 Problem

The `poll()` and `select()` interfaces suffer from the deficiency that the application must pass in an entire list of descriptors to be monitored, for every call. This has an immediate consequence of forcing the system to perform two memory copies across the user/kernel boundary, reducing the amount of memory bandwidth available for other activities. For large lists containing many thousands of descriptors, practical experience has shown that typically only a few hundred actually have any activity, making 95% of the copies unnecessary.

Upon return, the application must walk the entire list

to find the descriptors that the kernel marked as having activity. Since the kernel knew which descriptors were active, this results in a duplication of work; the application must recalculate the information that the system was already aware of. It would appear to be more efficient to have the kernel simply pass back a list of descriptors that it knows is active. Walking the list is an $O(N)$ activity, which does not scale well as N gets large.

Within the kernel, the situation is also not ideal. Space must be found to hold the descriptor list; for large lists, this is done by calling `malloc()`, and the area must in turn be freed before returning. After the copy is performed, the kernel must examine every entry to determine whether there is pending activity on the descriptor. If the kernel has not found any active descriptors in the current scan, it will then update the descriptor's `selinfo` entry; this information is used to perform a wakeup on the process in the event that it calls `tsleep()` while waiting for activity on the descriptor. After the process is woken up, it scans the list again, looking for descriptors that are now active.

This leads to 3 passes over the descriptor list in the case where `poll` or `select` actually sleep; once to walk the list in order to look for pending events and record the select information, a second time to find the descriptors whose activity caused a wakeup, and a third time in user space where the user walks the list to find the descriptors which were marked active by the kernel.

These problems stem from the fact that `poll()` and `select()` are stateless by design; that is, the kernel does not keep any record of what the application is interested in between system calls and must recalculate it every time. This design decision not to keep any state in the kernel leads to main inefficiency in the current implementation. If the kernel was able to keep track of exactly which descriptors the application was interested in, and only return a subset of these activated descriptors, much of the overhead could be eliminated.

3 Design Goals

When designing a replacement facility, the primary goal was to create a system that would be efficient and scalable to a large number of descriptors, on the order of several thousand. The secondary goal was to make the system flexible. UNIX based machines have traditionally lacked a robust facility for event notification. The `poll` and `select` interfaces are limited to socket and pipe descriptors; the user is unable to wait for other types of events, like file creation or deletion. Other events require the user to use a different interface; notably `siginfo` and family must be used to obtain notification of signal events, and calls to `aiowait` are needed to discover if an AIO call has completed.

Another goal was to keep the interface simple enough that it could be easily understood, and also possible to convert `poll()` or `select()` based applications to the new API with a minimum of changes. It was recognized that if the new interface was radically different, then it would essentially preclude modification of legacy applications which might otherwise take advantage of the new API.

Expanding the amount information returned to the application to more than just the fact that an event occurred was also considered desirable. For readable sockets, the user may want to know how many bytes are actually pending in the socket buffer in order to avoid multiple `read()` calls. For listening sockets, the application might check the size of the listen backlog in order to adapt to the offered load. The goal of providing more information was kept in mind when designing the new facility.

The mechanism should also be *reliable*, in that it should never silently fail or return an inconsistent state to the user. This goal implies that there should not be any fixed size lists, as they might overflow, and that any memory allocation must be done at the time of the system call, rather when activity occurs, to avoid losing events due to low memory conditions.

As an example, consider the case where several network packets arrive for a socket. We could consider each incoming packet as a discrete event, recording one event for each packet. However, the number of incoming packets is essentially unbounded, while the amount of memory in the system is finite; we would be unable to provide a guarantee that no events would be lost.

The result of the above scenario is that multiple packets are coalesced into a single event. Events that are delivered to the application may correspond to multiple occurrences of activity on the event source being monitored.

In addition, suppose a packet arrives containing N bytes, and the application, after receiving notification of the event, reads R bytes from the socket, where $R < N$. The next time the event API is called, there would be no notification of the $(N - R)$ bytes still pending in the socket buffer, because events would be defined in terms of arriving packets. This forces the application to perform extra bookkeeping in order to insure that it does not mistakenly lose data. This additional burden imposed on the application conflicts with the goal of providing a simple interface, and so leads to the following design decision.

Events will normally considered to be "level-triggered", as opposed to "edge-triggered". Another way of putting this is to say that an event is reported as long as a specified condition holds, rather than when activity is actually detected from the event source. The given condition could be as simple as "there is unread data in the buffer", or it could be more complex. This approach

handles the scenario described above, and allows the application to perform a partial read on a buffer, yet still be notified of an event the next time it calls the API. This corresponds to the existing semantics provided by poll() and select().

A final design criteria was that the API should be *correct*, in that events should only be reported if they are applicable. Consider the case where a packet arrives on a socket, in turn generating an event. However, before the application is notified of this pending event, it performs a close() on the socket. Since the socket is no longer open, the event should not be delivered to the application, as it is no longer relevant. Furthermore, if the event happens to be identified by the file descriptor, and another descriptor is created with the same identity, the event should be removed, to preclude the possibility of false notification on the wrong descriptor.

The correctness requirement should also extend to pre-existing conditions, where the event source generates an event prior to the application registering its interest with the API. This eliminates the race condition where data could be pending in a socket buffer at the time that the application registers its interest in the socket. The mechanism should recognize that the pending data satisfies the “level-trigger” requirement and create an event based on this information.

Finally, the last design goal for the API is that it should be possible for a library to use the mechanism without fear of conflicts with the main program. This allows 3rd party code that uses the API to be linked into the application without conflict. While on the surface this appears to be obvious, several counter examples exist. Within a process, a signal may only have a single signal handler registered, so library code typically can not use signals. X-window applications only allow for a single event loop. The existing select() and poll() calls do not have this problem, since they are stateless, but our new API, which moves some state into the kernel, must be able to have multiple event notification channels per process.

4 Kqueue API

The kqueue API introduces two new system calls outlined in Figure 1. The first creates a new kqueue, which is a notification channel, or queue, where the application registers which events it is interested in, and where it retrieves the events from the kernel. The returned value from kqueue() is treated as an ordinary descriptor, and can in turn be passed to poll(), select(), or even registered in another kqueue.

The second call is used by the application both to register new events with the kqueue, and to retrieve any pending events. By combining the registration and re-

```
int
kqueue(void)

int
kevent(int kq,
        const struct kevent *changelist, int nchanges,
        struct kevent *eventlist, int nevents,
        const struct timespec *timeout)

struct kevent {
    uintptr_t  ident;    // identifier for event
    short     filter;    // filter for event
    u_short   flags;     // action flags for kq
    u_int     fflags;    // filter flag value
    intptr_t  data;     // filter data value
    void      *udata;    // opaque identifier
}

EV_SET(&kev, ident, filter, flags, fflags, data, udata)
```

Figure 1: Kqueue API

trieval process, the number of system calls needed is reduced. Changes that should be applied to the kqueue are given in the *changelist*, and any returned events are placed in the *eventlist*, up to the maximum size allowed by *nevents*. The number of entries actually placed in the *eventlist* is returned by the kevent() call. The *timeout* parameter behaves in the same way as poll(); a zero-valued structure will check for pending events without sleeping, while a NULL value will block until woken up or an event is ready. An application may choose to separate the registration and retrieval calls by passing in a value of zero for *nchanges* or *nevents*, as appropriate.

Events are registered with the system by the application via a *struct kevent*, and an event is uniquely identified within the system by a *< kq, ident, filter >* tuple. In practical terms, this means that there can be only one *< ident, filter >* pair for a given kqueue.

The *filter* parameter is an identifier for a small piece of kernel code which is executed when there is activity from an event source, and is responsible for determining whether an event should be returned to the application or not. The interpretation of the *ident*, *fflags*, and *data* fields depend on which filter is being used to express the event. The current list of filters and their arguments are presented in the kqueue filter section.

The *flags* field is used to express what action should be taken on the kevent when it is registered with the system, and is also used to return filter-independent status information upon return. The valid flag bits are given in Figure 2.

The *udata* field is passed in and out of the kernel unchanged, and is not used in any way. The usage of this field is entirely application dependent, and is provided as a way to efficiently implement a function dispatch routine, or otherwise add an application identifier to the

Input flags:

- EV_ADD** Adds the event to the kqueue
- EV_ENABLE** Permit `kevent()` to return the event if it is triggered.
- EV_DISABLE** Disable the event so `kevent()` will not return it. The filter itself is not disabled.
- EV_DELETE** Removes the event from the kqueue. Events which are attached to file descriptors are automatically deleted when the descriptor is closed.
- EV_CLEAR** After the event is retrieved by the user, its state is reset. This is useful for filters which report state transitions instead of the current state. Note that some filters may automatically set this flag internally.
- EV_ONESHOT** Causes the event to return only the first occurrence of the filter being triggered. After the user retrieves the event from the kqueue, it is deleted.

Output flags:

- EV_EOF** Filters may set this flag to indicate filter-specific EOF conditions.
- EV_ERROR** If an error occurs when processing the changelist, this flag will be set.

Figure 2: Flag values for struct `kevent`

`kevent` structure.

4.1 Kqueue filters

The design of the kqueue system is based on the notion of filters, which are responsible for determining whether an event has occurred or not, and may also record extra information to be passed back to the user. The interpretation of certain fields in the `kevent` structure depends on which filter is being used. The current implementation comes with a few general purpose event filters, which are suitable for most purposes. These filters include:

- EVFILT_READ
- EVFILT_WRITE
- EVFILT_AIO
- EVFILT_VNODE
- EVFILT_PROC
- EVFILT_SIGNAL

The READ and WRITE filters are intended to work on any file descriptor, and the *ident* field contains the descriptor number. These filters closely mirror the behavior of `poll()` or `select()`, in that they are intended to return whenever there is data ready to read, or if the application can write without blocking. The kernel function corresponding to the filter depends on the descriptor type, so the implementation is tailored for the requirements of each type of descriptor in use. In general, the amount of data that is ready to read (or able to be written) will be returned in the *data* field within the `kevent` structure, where the application is free to use this information in whatever manner it desires. If the underlying descriptor supports a concept of EOF, then the EV_EOF flag will be set in the flags word structure as soon as it is detected, regardless of whether there is still data left for the application to read.

For example, the read filter for socket descriptors is triggered as long as there is data in the socket buffer greater than the SO_LOWAT mark, or when the socket has shutdown and is unable to receive any more data. The filter will return the number of bytes pending in the socket buffer, as well as set an EOF flag for the shutdown case. This provides more information that the application can use while processing the event. As EOF is explicitly returned when the socket is shutdown, the application no longer needs to make an additional call to `read()` in order to discover an EOF condition.

A non kqueue-aware application using the asynchronous I/O (aio) facility starts an I/O request by issuing `aio_read()` or `aio_write()`. The request then proceeds independently of the application, which must call `aio_error()` repeatedly to check whether the request has completed, and then eventually call `aio_return()` to collect the completion status of the request. The AIO filter replaces this polling model by allowing the user to register the aio request with a specified kqueue at the time the I/O request is issued, and an event is returned under the same conditions when `aio_error()` would successfully return. This allows the application to issue an `aio_read()` call, proceed with the main event loop, and then call `aio_return()` when the `kevent` corresponding to the aio is returned from the kqueue, saving several system calls in the process.

The SIGNAL filter is intended to work alongside the normal signal handling machinery, providing an alternate method of signal delivery. The *ident* field is interpreted as a signal number, and on return, the *data* field contains a count of how often the signal was sent to the application. This filter makes use of the EV_CLEAR flag internally, by clearing its state (count of signal occurrence) after the application receives the event notification.

The VNODE filter is intended to allow the user to register an interest in changes that happen within the filesystem. Accordingly, the *ident* field should contain a de-

Input/Output Flags:

NOTE_EXIT Process exited.

NOTE_FORK Process called fork()

NOTE_EXEC Process executed a new process via `execve(2)` or similar call.

NOTE_TRACK Follow a process across fork() calls. The parent process will return with **NOTE_TRACK** set in the flags field, while the child process will return with **NOTE_CHILD** set in fflags and the parent PID in data.

Output Flags only:

NOTE_CHILD This is the child process of a TRACKed process which called fork().

NOTE_TRACKERR This flag is returned if the system was unable to attach an event to the child process, usually due to resource limitations.

Figure 3: Flags for EVFILT_PROC

descriptor corresponding to an open file or directory. The *fflags* field is used to specify which actions on the descriptor the application is interested in on registration, and upon return, which actions have occurred. The possible actions are:

NOTE_DELETE
NOTE_WRITE
NOTE_EXTEND
NOTE_ATTRIB
NOTE_LINK
NOTE_RENAME

These correspond to the actions that the filesystem performs on the file and thus will not be explained here. These notes may be OR-d together in the returned kevent, if multiple actions have occurred. E.g.: a file was written, then renamed.

The final general purpose filter is the PROC filter, which detects process changes. For this filter, the *ident* field is interpreted as a process identifier. This filter can watch for several types of events, and the *fflags* that control this filter are outlined in Figure 3.

5 Usage and Examples

Kqueue is designed to reduce the overhead incurred by `poll()` and `select()`, by efficiently notifying the user of

an event that needs attention, while also providing as much information about that event as possible. However, kqueue is not designed to be a drop in replacement for `poll`; in order to get the greatest benefits from the system, existing applications will need to be rewritten to take advantage of the unique interface that kqueue provides.

A traditional application built around `poll` will have a single structure containing all active descriptors, which is passed to the kernel every time the application goes through the central event loop. A kqueue-aware application will need to notify the kernel of any changes to the list of active descriptors, instead of passing in the entire list. This can be done either by calling `kevent()` for each update to the active descriptor list, or by building up a list of descriptor changes and then passing this list to the kernel the next time the event loop is called. The latter approach offers better performance, as it reduces the number of system calls made.

While the previous API section for kqueue may appear to be complex at first, much of the complexity stems from the fact that there are multiple event sources and multiple filters. A program which only wants READ/WRITE events is actually fairly simple. Examples on the following pages illustrate how a program using `poll()` can be easily converted to use `kqueue()` and also presents several code fragments illustrating the use of the other filters.

The code in Figure 4 illustrates typical usage of the `poll()` system call, while the code in Figure 5 is a line-by-line conversion of the same code to use kqueue. While admittedly this is a simplified example, the mapping between the two calls is fairly straightforward. The main stumbling block to a conversion may be the lack of a function equivalent to `update_fd`, which makes changes to the array containing the `pollfd` or `kevent` structures.

If the *udata* field is initialized to the correct function prior to registering a new kevent, it is possible to simplify the dispatch loop even more, as shown in Figure 6.

Figure 7 contains a fragment of code that illustrates how to have a signal event delivered to the application. Note the call to `signal()` which establishes a NULL signal handler. Prior to this call, the default action for the signal is to terminate the process. Ignoring the signal simply means that no signal handler will be called after the signal is delivered to the process.

Figure 8 presents code that monitors a descriptor corresponding to a file on an ufs filesystem for specified changes. Note the use of **EV_CLEAR**, which resets the event after it is returned; without this flag, the event would be repeatedly returned.

The behavior of the PROC filter is best illustrated with the example below. A PROC filter may be attached to any process in the system that the application can see, it is not limited to its descendants. The filter may attach to a privileged process; there are no security implications, as

```

handle_events()
{
    int i, n, timeout = TIMEOUT;

    n = poll(pfd, nfds, timeout);

    if (n <= 0)
        goto error_or_timeout;
    for (i = 0; n != 0; i++) {
        if (pfd[i].revents == 0)
            continue;
        n--;
        if (pfd[i].revents &
            (POLLERR | POLLNVAL))
            /* error */
        if (pfd[i].revents & POLLIN)
            readable_fd(pfd[i].fd);
        if (pfd[i].revents & POLLOUT)
            writeable_fd(pfd[i].fd);
    }
    ...
}

update_fd(int fd, int action,
          int events)
{
    if (action == ADD) {
        pfd[fd].fd = fd;
        pfd[fd].events = events;
    } else
        pfd[fd].fd = -1;
}

```

Figure 4: Original poll() code

all information can be obtained through 'ps'. The term 'see' is specific to FreeBSD's jail code, which isolates certain groups of processes from each other.

There is single notification for each fork(), if the FORK flag is set in the process filter. If the TRACK flag is set, then the filter actually creates and registers a new knote, which is in turn attached to the new process. This new knote is immediately activated, with the CHILD flag set.

The fork functionality was added in order to trace the process's execution. For example, suppose that an EVFILT_PROC filter with the flags (FORK, TRACK, EXEC, EXIT) is registered for process A, which then forks off two children, processes B & C. Process C then immediately forks off another process D, which calls exec() to run another program, which in turn exits. If the application was to call kevent() at this point, it would find 4 kevents waiting:

```

ident: A, fflags: FORK
ident: B, fflags: CHILD      data: A
ident: C, fflags: CHILD, FORK data: A
ident: D, fflags: CHILD, EXEC, EXIT data: C

```

The knote attached to the child is responsible for re-

```

handle_events()
{
    int i, n;
    struct timespec timeout =
        { TMOUT_SEC, TMOUT_NSEC };

    n = kevent(kq, ch, nchanges,
               ev, nevents, &timeout);
    if (n <= 0)
        goto error_or_timeout;
    for (i = 0; i < n; i++) {

        if (ev[i].flags & EV_ERROR)
            /* error */

        if (ev[i].filter == EVFILT_READ)
            readable_fd(ev[i].ident);
        if (ev[i].filter == EVFILT_WRITE)
            writeable_fd(ev[i].ident);
    }
    ...
}

update_fd(int fd, int action,
          int filter)
{
    EV_SET(&ch[nchanges], fd, filter,
           action == ADD ? EV_ADD
                        : EV_DELETE,
           0, 0, 0);
    nchanges++;
}

```

Figure 5: Direct conversion to kevent()

turning mapping between the parent and child process ids.

6 Implementation

The focus of activity in the Kqueue system centers on a data structure called a knote, which directly corresponds to the kevent structure seen by the application. The knote ties together the data structure being monitored, the filter used to evaluate the activity, the kqueue that it is on, and links to other knotes. The other main data structure is the kqueue itself, which serves a twofold purpose: to provide a queue containing knotes which are ready to deliver to the application, and to keep track of the knotes which correspond to the kevents the application has registered its interest in. These goals are accomplished by the use of three sub data structures attached to the kqueue:

1. A list for the queue itself, containing knotes that have previously been marked active.
2. A small hash table used to look up knotes whose ident field does not correspond to a descriptor.


```

int i, n;
struct timespec timeout =
    { TMOUT_SEC, TMOUT_NSEC };
void (* fcn)(struct kevent *);

n = kevent(kq, ch, nchanges,
    ev, nevents, &timeout);
if (n <= 0)
    goto error_or_timeout;
for (i = 0; i < n; i++) {
    if (ev[i].flags & EV_ERROR)
        /* error */
        fcn = ev[i].udata;
        fcn(&ev[i]);
}

```

Figure 6: Using udata for direct function dispatch

```

struct kevent ev;
struct timespec nullts = { 0, 0 };

EV_SET(&ev, SIGHUP, EVFILT_SIGNAL,
    EV_ADD | EV_ENABLE, 0, 0, 0);
kevent(kq, &ev, 1, NULL, 0, &nullts);

signal(SIGHUP, SIG_IGN);
for (;;) {
    n = kevent(kq, NULL, 0, &ev, 1, NULL);
    if (n > 0)
        printf("signal %d delivered"
            " %d times\n",
            ev.ident, ev.data);
}

```

Figure 7: Using kevent for signal delivery

```

struct kevent ev;
struct timespec nullts = { 0, 0 };

EV_SET(&ev, fd, EVFILT_VNODE,
    EV_ADD | EV_ENABLE | EV_CLEAR,
    NOTE_RENAME | NOTE_WRITE |
    NOTE_DELETE | NOTE_ATTRIB, 0, 0);
kevent(kq, &ev, 1, NULL, 0, &nullts);

for (;;) {
    n = kevent(kq, NULL, 0, &ev, 1, NULL);

    if (n > 0) {
        printf("The file was");
        if (ev.fflags & NOTE_RENAME)
            printf(" renamed");
        if (ev.fflags & NOTE_WRITE)
            printf(" written");
        if (ev.fflags & NOTE_DELETE)
            printf(" deleted");
        if (ev.fflags & NOTE_ATTRIB)
            printf(" chmod/chowned");
        printf("\n");
    }
}

```

Figure 8: Using kevent to watch for file changes

3. A linear array of singly linked lists indexed by descriptor, which is allocated in exactly the same fashion as a process' open file table.

The hash table and array are lazily allocated, and the array expands as needed according to the largest file descriptor seen. The kqueue must record all knots that have been registered with it in order to destroy them when the kq is closed by the application. In addition, the descriptor array is used when the application closes a specific file descriptor, in order to delete any knots corresponding with the descriptor. An example of the links between the data structures is shown below.

6.1 Registration

Initially, the application calls `kqueue()` to allocate a new kqueue (henceforth referred to as kq). This involves allocation of a new descriptor, a struct kqueue, and entry for this structure in the open file table. Space for the array and hash tables are not initialized at this time.

The application then calls `kevent()`, passing in a pointer to the changelist that should be applied. The kevents in the changelist are copied into the kernel in chunks, and then each one is passed to `kqueue_register()` for entry into the kq. The `kqueue_register()` function uses the `<ident, filter>` pair to lookup a matching knot attached to the kq. If no knot is found, a new one may be allocated if the `EV_ADD` flag is set. The knot is initialized from the kevent structure passed in, then the filter attach routine (detailed below) is called to attach the knot to the event source. Afterwards, the new knot is linked to either the array or hash table within the kq. If an error occurs while processing the changelist, the kevent that caused the error is copied over to the eventlist for return to the application. Only after the entire changelist is processed does `kqueue_scan()` called in order to dequeue events for the application. The operation of this routine is detailed in the Delivery section.

6.2 Filters

Each filter provides a vector consisting of three functions: `{attach, detach, filter}`. The attach routine is responsible for attaching the knot to a linked list within the structure which receives the events being monitored, while the detach routine is used to remove the knot this list. These routines are needed because the locking requirements and location of the attachment point are different for each data structure.

The filter routine is called when there is any activity from the event source, and is responsible for deciding whether the activity satisfies a condition that would cause an event to be reported to the application. The specifics

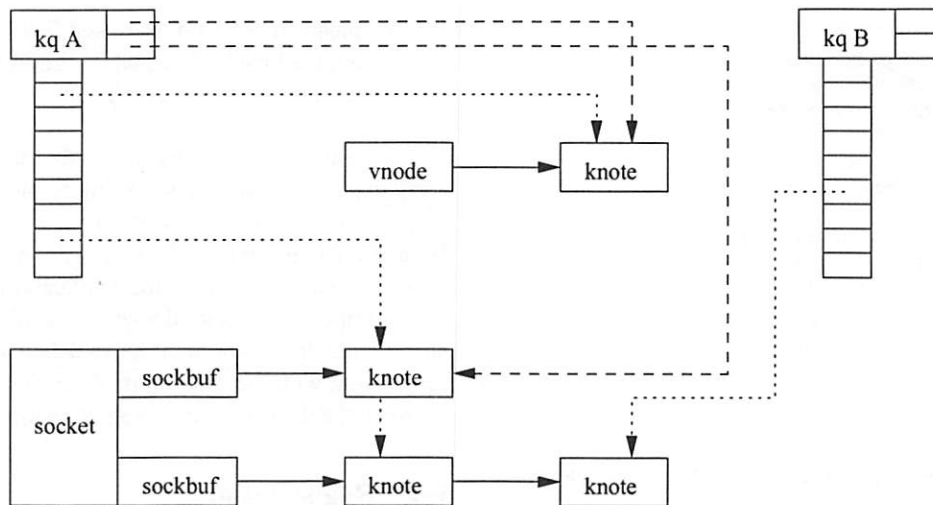


Figure 9: Two kqueues, their descriptor arrays, and active lists. Note that kq A has two knotes queued in its active list, while kq B has none. The socket has a klist for each sockbuf, and as shown, knotes on a klist may belong to different kqueues.

of the condition are encoded within the filter, and thus are dependent on which filter is used, but normally correspond to specific states, such as whether there is data in the buffer, or if an error has been observed. The filter must return a boolean value indicating whether an event should be delivered to the application. It may also perform some “side effects” if it chooses by manipulating the *flag* and *data* values within the knote. These side effects may range from merely recording the number of times the filter routine was called, or having the filter copy extra information out to user space.

All three routines completely encapsulate the information required to manipulate the event source. No other code in the kqueue system is aware of where the activity comes from or what an event represents, other than asking the filter whether this knote should be activated or not. This simple encapsulation is what allows the system to be extended to other event sources simply by adding new filters.

6.3 Activity on Event Source

When activity occurs (a packet arrives, a file is modified, a process exits), a data structure is typically modified in response. Within the code path where this happens, a hook is placed for the kqueue system, this takes the form of a *knote()* call. This function takes a singly linked list of knotes (unimaginatively referred to here as a klist) as an argument, along with an optional hint for the filter. The *knote()* function then walks the klist making calls to the filter routine for each knote. As the knote contains a reference to the data structure that it is attached to, the filter may choose to examine the data structure in deciding

whether an event should be reported. The hint is used to pass in additional information, which may not be present in the data structure the filter examines.

If the filter decides the event should be returned, it returns a truth value and the *knote()* routine links the knote onto the tail end of the active list in its corresponding kqueue, for the application to retrieve. If the knote is already on the active list, no action is taken, but the call to the filter occurs in order to provide an opportunity for the filter to record the activity.

6.4 Delivery

When *kqueue_scan()* is called, it appends a special knote marker at the end of the active list, which bounds the amount of work that should be done; if this marker is dequeued while walking the list, it indicates that the scan is complete. A knote is then removed from the active list, and the flags field is checked for the *EV_ONESHOT* flag. If this is not set, then the filter is called again with a query hint; this gives the filter a chance to confirm that the event is still valid, and insures correctness. The rationale for this is the case where data arrives for a socket, which causes the knote to be queued, but the application happens to call *read()* and empty the socket buffer before calling *kevent*. If the knote was still queued, then an event would be returned telling the application to read an empty buffer. Checking with the filter at the time the event is dequeued, assures us that the information is up to date. It may also be worth noting that if a pending event is deactivated via *EV_DISABLE*, its removal from the active queue is delayed until this point.

Information from the knote is then copied into a *kevent*

structure within the event list for return to the application. If `EV_ONESHOT` is set, then the knote is deleted and removed from the `kq`. Otherwise if the filter indicates that the event is still active and `EV_CLEAR` is not set, then the knote is placed back at the tail of the active list. The knote will not be examined again until the next scan, since it is now behind the marker which will terminate the scan. Operation continues until either the marker is dequeued, or there is no more space in the eventlist, at which time the marker is forcibly dequeued, and the routine returns.

6.5 Miscellaneous Notes

Since an ordinary file descriptor references the `kqueue`, it can take part in any operations that normally can be performed on a descriptor. The application may `select()`, `poll()`, `close()`, or even create a `kevent` referencing a `kqueue`; in these cases, an event is delivered when there is a knote queued on the active list. The ability to monitor a `kqueue` from another `kqueue` allows an application to implement a priority hierarchy by choosing which `kqueue` to service first.

The current implementation does not pass `kqueue` descriptors to children unless the new child will share its file table with the parent via `rfork(RFFDG)`. This may be viewed as an implementation detail; fixing this involves making a copy of all knote structures at `fork()` time, or marking them as copy on write.

Knotes are attached to the data structure they are monitoring via a linked list, contrasting with the behavior of `poll()` and `select()`, which record a single `pid` within the `selinfo` structure. While this may be a natural outcome from the way knotes are implemented, it also means that the `kqueue` system is not susceptible to select collisions. As each knote is queued in the active list, only processes sleeping on that `kqueue` are woken up.

As hints are passed to all filters on a `klist`, regardless of type, when a single `klist` contains multiple event types, care must be taken to insure that the hint uniquely identifies the activity to the filters. An example of this may be seen in the `PROC` and `SIGNAL` filters. These share the same `klist`, hung off of the process structure, where the hint value is used to determine whether the activity is signal or process related.

Each `kevent` that is submitted to the system is copied into kernel space, and events that are dequeued are copied back out to the eventlist in user space. While adding slightly more copy overhead, this approach was preferred over an AIO style solution where the kernel directly updates the status of a control block that is kept in user space. The rationale for this was that it would be easier for the user to find and resolve bugs in the application if the kernel is not allowed to write directly to lo-

cations in user space which the user could possibly have freed and reused by accident. This has turned out to have an additional benefit, as applications may choose to “fire and forget” by submitting an event to the kernel and not keeping additional state around.

7 Performance

Measurements for performance numbers in this section were taken on a Dell PowerEdge 2300 equipped with an Intel Pentium-III 600Mhz CPU and 512MB memory, running FreeBSD 4.3-RC.

The first experiment was to determine the costs associated with the `kqueue` system itself. For this a program similar to `lmbench` [6] was used. The command under test was executed in a loop, with timing measurements taken outside the loop, and then averaged by the number of loops made. Times were measured using the `clock_gettime(CLOCK_REALTIME)` facility provided by FreeBSD, which on the platform under test has a resolution of 838 nanoseconds. Time required to execute the loop itself and the system calls to `clock_gettime()` were measured and the reported values for the final times were adjusted to eliminate the overhead. Each test was run 1024 times, with the first test not included in the measurements, in order to eliminate adverse cold cache effects. The mean value of the tests were taken; in all cases, the difference between the mean and median is less than one standard deviation.

In the first experiment, a varying number of sockets or files were created, and then passed to `kevent` or `poll`. The time required for the call to complete was recorded, and no activity was pending on any of the descriptors. For both system calls, this measures the overhead needed to copy the descriptor sets, and query each descriptor for activity. For the `kevent` system call, this also reflects the overhead needed to establish the internal knote data structure.

As shown in Figure 10, it takes twice as long to add a new knote to a `kqueue` as opposed to calling `poll`. This implies that for applications that `poll` a descriptor exactly once, `kevent` will not provide a performance gain, due to the amount of overhead required to set up the knote linkages. The differing results between the socket and file descriptors reflects the different code paths used to check activity on different file types in the system.

After the initial `EV_ADD` call to add the descriptors to the `kqueue`, the time required to check these descriptors was recorded; this is shown in the “`kq_descriptor`” line in the graph above. In this case, there was no difference between file types. In all cases, the time is constant, since there is no activity on any of the registered descriptors.

This provides a lower bound on the time required for a given `kevent` call, regardless of the number of descriptors

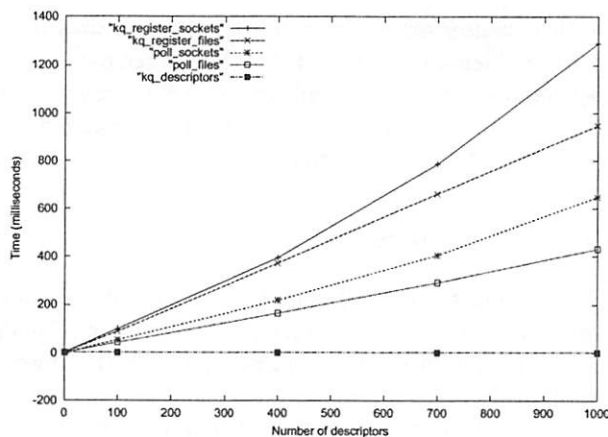


Figure 10: Time needed for initial kqueue call. Note y-axis origin is shifted in order to better see kqueue results.

that are being monitored.

The main cost associated with the kevent call is the process of registering a new knot with the system; however, once this is done, there is negligible cost for monitoring the descriptor if it is inactive. This contrasts with poll, which incurs the same cost regardless of whether the descriptor is active or inactive.

The upper bound on the time needed for a kevent call after the descriptors are registered would be if every single descriptor was active. In this case the kernel would have to do the maximum amount of work by checking each descriptor's filter for validity, and then returning every kevent in the kqueue to the user. The results of this test are shown in Figure 11, with the poll values reproduced again for comparison.

In this graph, the lines for kqueue are worst case times; in which every single descriptor is found to be active. The best case time is near zero, as given by the earlier "kq_descriptor" line. In an actual workload, the actual time is somewhere inbetween, but in either case, the total time taken is less than that for poll().

As evidenced by the two graphs above, the amount of time saved by kqueue over poll depends on the number of times that a descriptor is monitored for an event, and the amount of activity that is present on a descriptor. Figure 12 shows accumulated time required to check a single descriptor for both kqueue and poll. The poll line is constant, while the two kqueue lines give the best and worst case scenarios for a descriptor. Times here are averaged from the 100 file descriptor case in the previous graphs. This graph shows that despite a higher startup time for kqueue, unless the descriptor is polled less than 4 times, kqueue has a lower overall cost than poll.

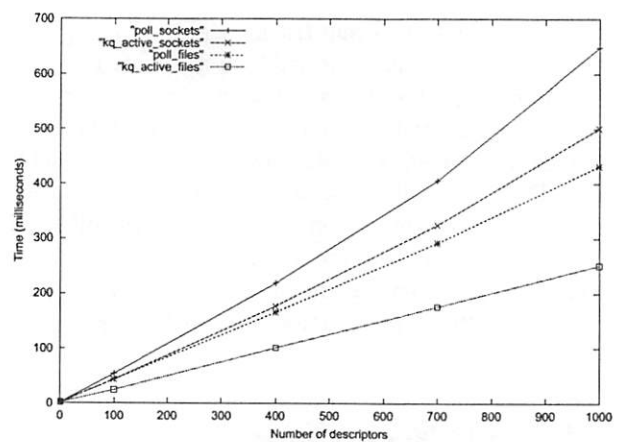


Figure 11: Time required when all descriptors are active.

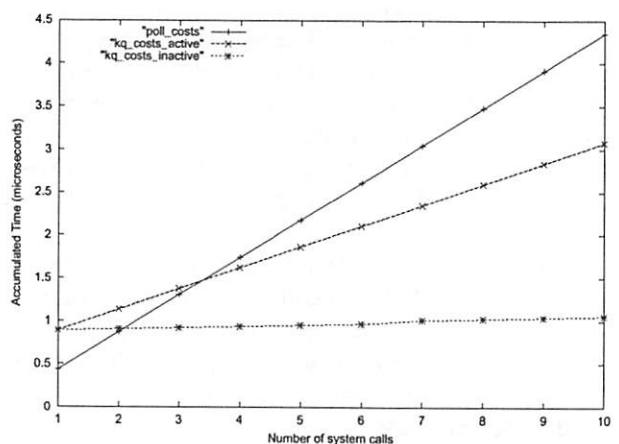


Figure 12: Accumulated time for kqueue vs poll

7.1 Individual operations

The state of kqueue is maintained by using the action field in the kevent to alter the state of the knots. Each of these actions takes a different amount of time to perform, as illustrated by Figure 13. These operations are performed on socket descriptors; the graphs for file descriptors (ttys) are similar. While enable/disable have a lower cost than add/delete, recall that this only affects returning the kevent to the user; the filter associated with the knot will still be executed.

7.2 Application level benchmarks

Web Proxy Cache

Two real-world applications were modified to use the kqueue system call; a commercial web caching proxy server, and the httpd [9] Web server. Both of these applications were run on the platform described earlier.

The client machine for running network tests was an Alpha 264DP, using a single 21264 EV6 666Mhz pro-

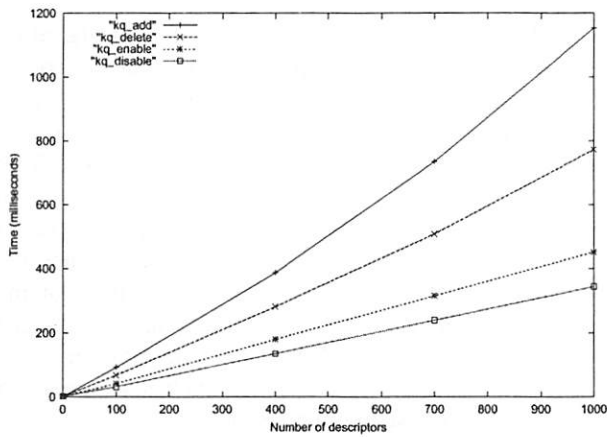


Figure 13: Time required for each kqueue operation.

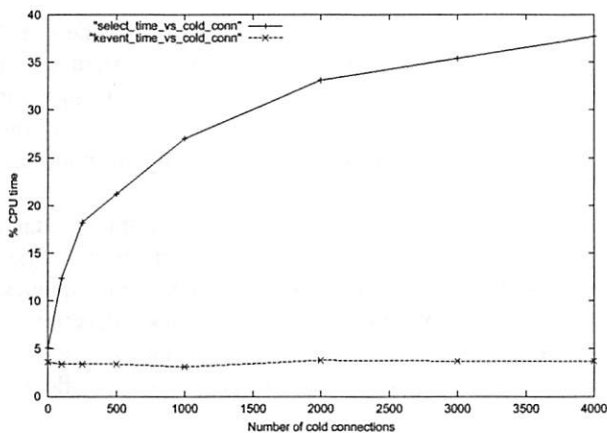


Figure 14: Kernel CPU time consumed by system call.

processor, and 512MB memory, running FreeBSD 4.3-RC. Both machines were equipped with a Netgear GA620 Gigabit Ethernet card, and connected via a Cisco Catalyst 3500 XL Gigabit switch. No other traffic was present on the switch at the time of the tests. For the web cache, all files were loaded into the cache from a web server before starting the test.

In order to generate a workload for the web proxy server with the equipment available, the `http_load` [8] tool was used. This was configured to request URLs from a set of 1000 1KB and 10 1MB cached documents from the proxy, while maintaining 100 parallel connections. Another program was used to keep a varying number of idle connections open to the server. This approach follows earlier research that shows that web servers have a small set of active connections, and a larger number of inactive connections [2]. Performance data for the tests were collected on the server system by running the kernel profiler (`kgmon`) while the cache was under load.

Figure 14 shows the amount of CPU time that each system call (and its direct descendants) use as the num-

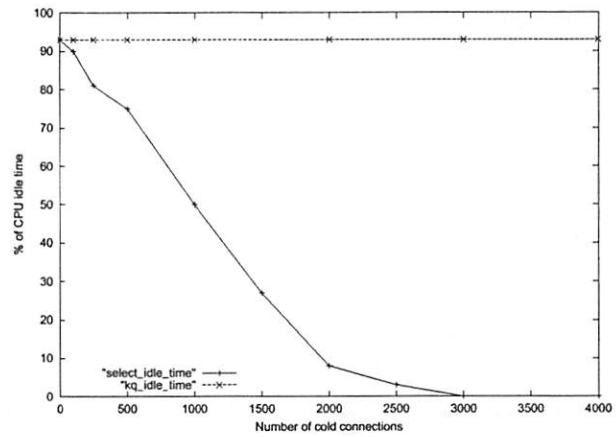


Figure 15: Amount of idle time remaining.

ber of active connections is held at 100, and the number of cold connections varies. Observing the graph, see that the kqueue times are constant regardless of the number of inactive connections, as expected from the microbenchmarks. The select based approach starts nearing the saturation point as the amount of idle CPU time decreases. Figure 15 shows a graph of idle CPU time, as measured by `vmstat`, and it can be seen that the system is essentially out of resources by the time there are 2000 connections.

thttpd Web Server

The `thttpd` Web Server [9] was modified to add kqueue support to its `fdwatch` descriptor management code, and the performance of the resulting server was compared to the original code.

For benchmarking the server, the `httperf` [7] measurement package was used. The size of the `FD_SETSIZE` array was increased in order to support more than 1024 open descriptors. The value of `net.inet.tcp.msl` on both client and server machines was decreased from 30 seconds to 5 seconds in order to recycle the network port space at a higher rate. After the server was started, and before any measurements were taken, a single dry run was done using the maximum number of idle connections between the client and server. Doing this allows the kernel portion of the webserver process to preallocate space for the open file descriptor kqueue descriptor tables, as well as allowing the user portion of the process to allocate the space needed for the data structures. If this was not done, the response rate as observed from the client varies as the process attempts to allocate memory.

The offered load from client using `httperf` was kept constant at 500 requests per second for this test, while the number of idle connections opened with `idletime` was varied. The result of the test is the reply time as reported by `httperf`. The reply rate for all tests was equal to the request rate, while the number of errors was negligible

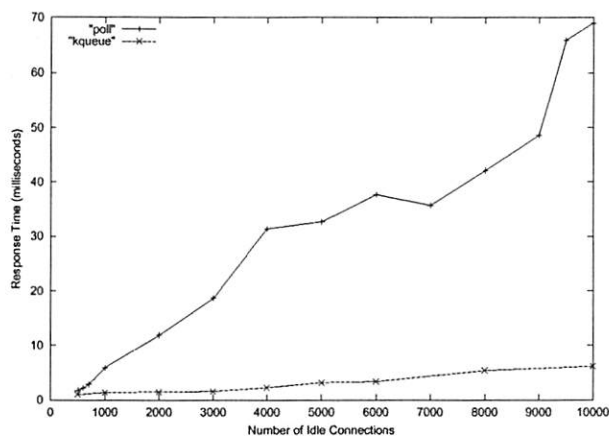


Figure 16: Response time from httperf

(< 2 in all cases).

The idle time on the server machine was monitored during the test using `vmstat`. The unmodified `thttpd` server runs out of `cpu` when the number of idle connections is around 600, while the modified server still has approximately 48% idle time with 10,000 idle connections.

8 Related Work

This section presents some of the other work done in this area.

POSIX signal queues

POSIX signal queues are part of the Single Unix Specification [5], and allow signals to be queued for delivery to an application, along with some additional information. For every event that should be delivered to the application, a signal (typically `SIGIO`) is generated, and a structure containing the file descriptor is allocated and placed on the queue for the signal handler to retrieve. No attempt at event aggregation is performed, so there is no fixed bound on the queue length for returned events. Most implementations silently bound the queue length to a fixed size, dropping events when the queue becomes too large. The structure allocation is performed when the event is delivered, opening up the possibility of losing events during a resource shortage.

The signal queues are stateless, so the application must handle the bookkeeping required to determine whether there is residual information left from the initial event. The application must also be prepared to handle stale events as well. As an example, consider what happens when a packet arrives, causing an event to be placed on a signal queue, and then dequeued by the signal handler. Before any additional processing can happen, a second

packet arrives and a second event is in turn placed on the signal queue. The application may, in the course of processing the first event, close the descriptor corresponding to the network channel that the packets are associated with. When the second event is retrieved from the signal queue, it is now “stale” in the sense that it no longer corresponds to an open file descriptor. What is worse, the descriptor may have been reused for another open file, resulting in a false reporting of activity on the new descriptor. A further drawback to the signal queue approach is that the use of signals as a notification mechanism precludes having multiple event handlers, making it unsuitable for use in library code.

get_next_event

This proposed API by Banga, Mogul and Druschel [2] motivated the author to implement system under FreeBSD that worked in a similar fashion, using their concept of hinting. The practical experience gained from real world usage of an application utilizing this approach inspired the concept of `kqueue`.

While the original system described by Banga, et.al., performs event coalescing, it also suffers from “stale” events, in the same fashion of POSIX signal queues. Their implementation is restricted to socket descriptors, and also uses a list of fixed size to hold hints, falling back to the behavior of a normal `select()` upon list overflow.

SGI's /dev/imon

`/dev/imon` [3] is an inode monitor, and where events within the filesystem are sent back to user-space. This is the only other interface that the author is aware of that is capable of performing similar operations as the `VNODE` filter. However, only a single process can read the device node at once; SGI handles this by creating a daemon process called `fmon` that the application may contact to request information from.

Sun's /dev/poll

This system [4] appears to come closest to the design outlined in this paper, but has some limitations as compared to `kqueue`. Applications are able to open `/dev/poll` to obtain a filedescriptor that behaves similarly to a `kq` descriptor. Events are passed to the kernel by performing a `write()` on the descriptor, and are read back via an `ioctl()` call. The returned information is limited to an `revent` field, similarly to that found in `poll()`, and the interface restricted to sockets; it cannot handle FIFO descriptors or other event sources (signals, filesystem events).

The interface also does not automatically handle the case where a descriptor is closed by the application, but

instead keeps returning POLLNVAL for that descriptor until removed from the interest set or reused by the application.

The descriptor obtained by opening `/dev/poll` can not in turn be selected on, precluding construction of hierarchical or prioritized queues. There is no equivalent to `kqueue`'s filters for extending the behavior of the system, nor support for direct function dispatch as there is with `kqueue`.

9 Conclusion

Applications handling a large number of events are dependent on the efficiency of event notification and delivery. This paper has presented the design criteria for a generic and scalable event notification facility, as well as an alternate API. This API was implemented in FreeBSD and committed to the main CVS tree in April 2000.

Overall, the system performs to expectations, and applications which previously found that select or poll was a bottleneck have seen performance gains from using `kqueue`. The author is aware of the system being used in several major applications such as web servers, web proxy servers, irc daemons, netnews transports, and mail servers, to name a few.

The implementation described here has been adopted by OpenBSD, and is in the process of being brought into NetBSD as well, so the API is not limited to a single operating system. While the measurements in this paper have concentrated primarily on the socket descriptors, other filters also provide performance gains.

The "tail -f" command in FreeBSD was historically implemented by stat'ing the file every 1/4 second in order to see if the file had changed. Replacing this polling approach with a `kq` `VNODE` filter provides the same functionality with less overhead, for those underlying filesystems that support `kqueue` event notification.

The AIO filter is used to notify the application when an AIO request is completed, enabling the main dispatch loop to be simplified to a single `kevent` call instead of a combination of `poll`, `aio_error`, and `aio_suspend` calls.

The DNS resolver library routines (`res_*`) used `select()` internally in order to wait for a response from the name server. On the FreeBSD project's heavily loaded e-mail exploder which uses postfix for mail delivery, the system was seeing an extremely high number of select collisions, which causes every process using `select()` to be woken up. Changing the resolver library to use `kqueue` was a successful example of using a private `kqueue` within a library routine, and also resulted in a performance gain by eliminating the select collisions.

The author is not aware of any other UNIX system which is capable of handling multiple event sources, nor one that can be trivially extended to handle additional

sources. Since the original implementation was released, the system has been extended down to the device layer, and now is capable of handling device-specific events as well. A device manager application is planned for this capability, where the user is notified of any change in hot-swappable devices in the system. Another filter that is in the process of being added is a `TIMER` filter which provides the application with as many oneshot or periodic timers as needed. Additionally, a high performance kernel audit trail facility may be implemented with `kqueue`, by having the user use a `kqueue` filter to selectively choose which auditing events should be recorded.

References

- [1] BANGA, G., AND MOGUL, J. C. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference* (New Orleans, LA, 1998).
- [2] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference* (1999), pp. 253–265.
- [3] `/dev/imon`. http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/a_man/cat7/imon.z.
- [4] `/dev/poll`. <http://docs.sun.com/ab2/coll.40.6/REFMAN7/@Ab2PageView/55123>.
- [5] GROUP, T. Single unix specification, 1997. <http://www.opengroup.org/online-pubs?DOC=007908799>.
- [6] MCVOY, L. W., AND STAELIN, C. Imbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference* (1996), pp. 279–294.
- [7] MOSBERGER, D., AND JIN, T. `httpperf`: A tool for measuring web server performance. In *First Workshop on Internet Server Performance* (June 1998), ACM, pp. 59–67.
- [8] POSKANZER, J. `http_load`. http://www.acme.com/software/http_load/.
- [9] POSKANZER, J. `thttpd`. <http://www.acme.com/software/thttpd/>.
- [10] PROVOS, N., AND LEVER, C. Scalable network i/o in linux, 2000.

Improving the FreeBSD SMP implementation

Greg Lehey
IBM LTC Ozlabs
grog@FreeBSD.org
grog@a1.ibm.com

ABSTRACT

UNIX-derived operating systems have traditionally have a simplistic approach to process synchronization which is unsuited to multiprocessor application. Initial FreeBSD SMP support kept this approach by allowing only one process to run in kernel mode at any time, and also blocked interrupts across multiple processors, causing seriously suboptimal performance of I/O bound systems. This paper describes work done to remove this bottleneck, replacing it with fine-grained locking. It derives from work done on BSD/OS and has many similarities with the approach taken in SunOS 5. Synchronization is performed primarily by a locking construct intermediate between a spin lock and a binary semaphore, termed *mutexes*. In general, mutexes attempt to block rather than to spin in cases where the likely wait time is long enough to warrant a process switch. The issue of blocking interrupt handlers is addressed by attaching a process context to the interrupt handlers. Despite this process context, an interrupt handler normally runs in the context of the interrupted process and is scheduled only when blocking is required.

Introduction

A crucial issue in the design of an operating system is the manner in which it shares resources such as memory, data structures and processor time. In the UNIX model, the main clients for resources are processes and interrupt handlers. Interrupt handlers operate completely in kernel space, primarily on behalf of the system. Processes normally run in one of two different modes, user mode and kernel mode. User mode code is the code of the program from which the process is derived, and kernel mode code is part of the kernel. This structure gives rise to multiple potential conflicts.

Use of processor time

The most obvious demand a process or interrupt routine places on the system is that it wants to run: it must execute instructions. In traditional UNIX, the rules governing this sharing are:

- There is only one processor. All code runs on it.
- If both an interrupt handler and a process are available to run, the interrupt handler runs.
- Interrupt handlers have different priorities. If one interrupt handler is running and one with a higher priority becomes runnable, the higher priority interrupt immediately preempts the lower priority interrupt.
- The scheduler runs when a process voluntarily relinquishes the processor, its time slice expires, or a higher-priority process becomes runnable. The scheduler chooses the highest priority process which is ready to run.
- If the process is in kernel mode when its time slice expires or a higher priority process becomes runnable, the system waits until it returns to user mode or sleeps before running the scheduler.

This method works acceptably for the single processor machines for which it was designed. In the following section, we'll see the reasoning behind

the last decision.

Kernel data objects

The most obvious problem is access to memory. Modern UNIX systems run with *memory protection*, which prevents processes in user mode from accessing the address space of other processes. This protection no longer applies in kernel mode: all processes share the kernel address space, and they need to access data shared between all processes. For example, the `fork()` system call needs to allocate a `proc` structure for the new process. The file `sys/kern_fork.c` contains the following code:

```
int
fork1(p1, flags, procp)
    struct proc *p1;
    int flags;
    struct proc **procp;
{
    struct proc *p2, *pptr;

    ...
    /* Allocate new proc. */
    newproc = zalloc(proc_zone);
```

The function `zalloc` takes a `struct proc` entry off a freelist and returns its address:

```
    item = z->zitems;
    z->zitems = ((void **) item)[0];
    ...
    return item;
```

What happens if the currently executing process is interrupted exactly between the first two lines of the code above, maybe because a higher priority process wants to run? `item` contains the pointer to the process structure, but `z->zitems` still points to it. If the interrupting code also allocates a process structure, it will go through the same code and return a pointer to the same memory area, creating the process equivalent of Siamese twins.

UNIX solves this issue with the rule “The UNIX kernel is non-preemptive”. This means that when a process is running in kernel mode, no other process can execute kernel code until the first process relinquishes the kernel voluntarily, either by returning to user mode, or by sleeping.

Synchronizing processes and interrupts

The non-preemption rule only applies to processes. Interrupts happen independently of process context, so a different method is needed. In de-

vice drivers, the process context (“top half”) and the interrupt context (“bottom half”) must share data. Two separate issues arise here: each half must ensure that any changes to shared data structures occur in a consistent manner, and they must find a way to synchronize with each other.

Protection

Each half must protect its data against change by the other half. For example, the buffer header structure contains a `flags` word with 32 flags, some set and reset by both halves. Setting and resetting bits requires multiple instructions on most architectures, so the potential for data corruption exists. UNIX solves this problem by locking out interrupts during critical sections. Top half code must explicitly lock out interrupts with the `spl` functions.¹ One of the most significant sources of bugs in drivers is inadequate synchronization with the bottom half.

Interrupt code does not need to perform any special synchronization: by definition, processes don’t run when interrupt code is active.

Blocking interrupts has a potential danger that an interrupt will not be serviced in a timely fashion. On PC hardware, this is particularly evident with serial I/O, which frequently generates an interrupt for every character. At 115200 bps, this equates to an interrupt every 85 μ s. In the past, this has given rise to the dreaded silo overflows; even on fast modern hardware it can be a problem. It’s also not easy to decide interrupt priorities: in the early days, disk I/O was given a high priority in order to avoid overruns, while serial I/O had a low priority. Nowadays disk controllers can handle transfers by themselves, but overruns are still a problem with serial I/O.

Waiting for the other half

In other cases, a process will need to wait for some event to complete. The most obvious example is I/O: a process issues an I/O request, and the driver initiates the transfer. It can be a long time before the transfer completes: if it’s reading

1. The naming goes back to the early days of UNIX on the PDP-11. The PDP-11 had a relatively simplistic level-based interrupt structure. When running at a specific level, only higher priority interrupts were allowed. UNIX named functions for setting the interrupt priority level after the PDP-11 `SPL` instruction, so initially the functions had names like `spl4` and `spl7`. Later machines came out with interrupt masks, and BSD changed the names to more descriptive names such as `splbio` (for block I/O) and `splhigh` (block out all interrupts).

keyboard input, for example, it could be weeks before the I/O completes. When the transfer completes, it causes an interrupt, so it's the interrupt handler which finally determines that the transfer is complete and notifies the process. Traditional UNIX performs this synchronization with the functions `sleep` and `wakeup`, though current BSD no longer uses `sleep`: it has been replaced with `tsleep`, which offers additional functionality.

The top half of a driver calls `sleep` or `tsleep` when it wants to wait for an event, and the bottom half calls `wakeup` when the event occurs. In more detail,

- The process issues a system call `read`, which brings it into kernel mode.
- `read` locates the driver for the device and calls it to initiate a transfer.
- `read` next calls `tsleep`, passing it the address of some unique object related to the request. `tsleep` stores the address in the proc structure, marks the process as sleeping and relinquishes the processor. At this point, the process is sleeping.
- At some later point, when the request is complete, the interrupt handler calls `wakeup` with the address which was passed to `tsleep`. `wakeup` runs through a list of sleeping processes and wakes all processes waiting on this particular address.

This method has problems even on single processors: the time to wake processes depends on the number of sleeping processes, which is usually only slightly less than the number of processes in the system. FreeBSD addresses this problem with 128 hashed sleep queues, effectively diminishing the search time by a factor of 128. A large system might have 10,000 processes running at the same time, so this is only a partial solution.

In addition, it is permissible for more than one process to wait on a specific address. In extreme cases dozens of processes wait on a specific address, but only one will be able to run when the resource becomes available; the rest call `tsleep` again. The term *thundering horde* has been devised to describe this situation. FreeBSD has partially solved this issue with the `wakeup_one` function, which only wakes the first process it finds. This still involves a linear search through a possibly large number of process structures, and it has the potential to deadlock if two unrelated

events map to the same address.

Adapting the UNIX model to SMP

A number of the basic assumptions of this model no longer apply to SMP, and others become more of a problem:

- More than one processor is available. Code can run in parallel.
- Interrupt handlers and user processes can run on different processors at the same time.
- The “non-preemption” rule is no longer sufficient to ensure that two processes can't execute at the same time, so it would theoretically be possible for two processes to allocate the same memory.
- Locking out interrupts must happen in every processor. This can adversely affect performance.

The initial FreeBSD model

The original version of FreeBSD SMP support solved these problems in a manner designed for reliability rather than performance: effectively it found a method to simulate the single-processor paradigm on multiple processors. Specifically, only one process could run in the kernel at any one time. The system ensured this with a spinlock, the so-called *Big Kernel Lock (BKL)*, which ensured that only one processor could be in the kernel at a time. On entry to the kernel, each processor attempted to get the BKL. If another processor was executing in kernel mode, the other processor performed a *busy wait* until the lock became free:

```
MPgetlock_edx:
1:      movl    (%edx), %eax
        movl    %eax, %ecx
        andl    $CPU_FIELD, %ecx
        cmpl    _cpu_lockid, %ecx
        jne     2f
        incl    %eax
        movl    %eax, (%edx)
        ret

2:      movl    $FREE_LOCK, %eax
        movl    _cpu_lockid, %ecx
        incl    %ecx
        lock
        cmpxchg %ecx, (%edx)
        jne     1b
        GRAB_HWI
        ret
```

In an extreme case, this waiting could degrade SMP performance to below that of a single processor machine.

How to solve the dilemma

Multiple processor machines have been around for a long time, since before UNIX was written. During this time, a number of solutions to this kind of problem have been devised. The problem was less to find a solution than to find a solution which would fit in the UNIX environment. At least the following synchronization primitives have been used in the past:

- *Counting semaphores* were originally designed to share a certain number of resources amongst potentially more consumers. To get access, a consumer decrements the semaphore counter, and when it is finished it increments it again. If the semaphore counter goes negative, the process is placed on a *sleep queue*. If it goes from -1 to 0, the first process on the sleep queue is activated. This approach is a possible alternative to `tsleep` and `wakeup` synchronization. In particular, it avoids a lengthy sequential search of sleeping processes.
- SunOS 5 uses *turnstiles* to address the sequential search problem in `tsleep` and `wakeup` synchronization. A turnstile is a separate queue associated with a specific wait address, so the need for a sequential search disappears.
- *Spin locks* have already been mentioned. FreeBSD used to spin indefinitely on the BKL, which doesn't make any sense, but they are useful in cases where the wait is short; a longer wait will result in a process being suspended and subsequently rescheduled. If the average wait for a resource is less than this time, then it makes sense to spin instead.
- *Blocking locks* are the alternative to spin locks when the wait is likely to be longer than it would take to reschedule. A typical implementation is similar to a counting semaphore with a count of 1.
- *Condition variables* are a kind of blocking lock where the lock is based on a condition, for example the absence of entries in a queue.

- *Read/write locks* address a different issue: frequently multiple processes may read specific data in parallel, but only one may write it.

There is some confusion in terminology with these locking primitives. In particular, the term *mutex* has been applied to nearly all of them at different times. We'll look at how FreeBSD uses the term in the next section.

One big problem with all locking primitives with the exception of spin locks is that they can block. This requires a process context: a UNIX interrupt handler can't block. This is one of the reasons that the old BKL was a spinlock, even though it could potentially use up most of processor time spinning.

The new FreeBSD implementation

The new implementation of SMP on FreeBSD bases heavily on the implementation in BSD/OS 5.0, which has not yet been released. Even the name *SMPng* ("new generation") was taken from BSD/OS. Due to the open source nature of FreeBSD, SMPng is available on FreeBSD before on BSD/OS.

The most radical difference in SMPng are:

- Interrupt code ("bottom half") now runs in a process context, enabling it to block if necessary. This process context is termed an *interrupt thread*.
- Interrupt lockout primitives (`splfoo`) have been removed. The low-level interrupt code still needs to block interrupts briefly, but the interrupt service routines themselves run with interrupts enabled. Instead of locking out interrupts, the system uses mutexes, which may be either spin locks or blocking locks.

Interrupt threads

The single most important aspect of the implementation is the introduction of a process or "thread" context for interrupt handlers. This change involves a number of tradeoffs:

- The process context allows a uniform approach to synchronization: it is no longer necessary to provide separate primitives to synchronize the top half and the bottom half. In particular, the *spl* primitives are no longer

needed. For compatibility reasons, the calls have been retained, but they translate to no-ops.

- The action of scheduling another process takes significantly longer than interrupt overhead, which also remains.
- The UNIX approach to scheduling does not allow preemption if the process is running in kernel mode.

SMPng solves the latency and scheduling issues with a technique known as *lazy scheduling*: on receiving an interrupt, the interrupt stubs note the PID of the interrupt thread, but they do not schedule the thread. Instead, it continues execution in the context of the interrupted process. The thread will be scheduled only in the following circumstances:

- If the thread has to block.
- If the interrupt nesting level gets too deep.

We expect this method to offer negligible overhead for the majority of interrupts.

From a scheduling viewpoint, the threads differ from normal processes in the following ways:

- They never enter user mode, so they do not have user text and data segments.
- They all share the address space of process 0, the swapper.
- They run at a higher priority than all user processes.
- Their priority is not adjusted based on load: it remains fixed.
- An additional process state `SWAIT` has been introduced for interrupt processes which are currently idle: the normal “idle” state is `SSLEEP`, which implies that the process is sleeping.

Experience with the BSD/OS implementation showed that the initial implementation of interrupt threads was a particularly error-prone process, and that the debugging tools were inadequate. Due to the nature of the FreeBSD project, we considered it imperative to have the system relatively functional at all times during the transition, so we decided to implement interrupt threads in two stages. The initial implementation was very similar to that of normal processes. This offered the benefits of relatively easy debugging and of stability, and the disadvantage of a significant

drop in performance: each interrupt could potentially cause two context switches, and the interrupt would not be handled while another process, even a user process, was in the kernel.

Experience with the initial implementation met expectations: we have seen no stability problems with the implementation, and the performance, though significantly worse, was not as bad as we had expected.

At the time of writing, we have improved the implementation somewhat by allowing limited kernel preemption, allowing interrupt threads to be scheduled immediately rather than having to wait for the current process to leave kernel mode. The potential exists for complete kernel preemption, where any higher priority process can preempt a lower priority process running in the kernel, but we are not sure that the benefits will outweigh the potential bug sources.

The final *lazy scheduling* implementation has been tested, but it is not currently in the -CURRENT kernel. Due to the current kernel lock implementation, it would not show any significant performance increase, and problems can be expected as additional kernel components are migrated from under `Giant`.

Not all interrupts have been changed to threaded interrupts. In particular, the old *fast interrupts* remain relatively unchanged, with the restriction that they may not use any blocking mutexes. Fast interrupts have typically been used for the serial drivers, and are specific to FreeBSD: BSD/OS has no corresponding functionality.

Locking constructs

The initial BSD/OS implementation defined two basic types of lock, called *mutex*:

- The default locking construct is the *spin/sleep mutex*. This is similar in concept to a semaphore with a count of 1, but the implementation allows spinning for a certain period of time if this appears to be of benefit (in other words, if it is likely that the mutex will become free in less time than it would take to schedule another process), though this feature is not currently in use. It also allows the user to specify that the mutex should *not* spin. If the process cannot obtain the mutex, it is placed on a sleep queue and woken when the resource becomes available.

- An alternate construct is a *spin mutex*. This corresponds to the spin lock which was already present in the system. Spin mutexes are used only in exceptional cases.

The implementation of these locks was derived almost directly from BSD/OS, but has since been modified significantly.

In addition to these locks, the FreeBSD project has included two further locking constructs:

Condition variables are built on top of mutexes. They consist of a mutex and a wait queue. The following operations are supported:

- Acquire a condition variable with `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()` or `cv_timedwait_sig()`.
- Before acquiring the condition variable, the associated mutex must be held. The mutex will be released before sleeping and reacquired on wakeup.
- Unblock one waiter with `cv_signal()`.
- Unblock all waiters with `cv_broadcast()`.
- Wait for queue empty with `cv_waitq_empty`.
- Same functionality available from the `msleep` function.

Shared/exclusive locks, or *sx locks*, are effectively read-write locks. The difference in terminology came from an intention to add additional functionality to these locks. This functionality has not been implemented, so currently *sx locks* are the same thing as read-write locks: they allow access by multiple readers or a single writer.

The implementation of *sx locks* is relatively expensive:

```
struct sx {
    struct lock_object sx_object;
    struct mtx      sx_lock;
    int             sx_cnt;
    struct cv       sx_shrd_cv;
    int             sx_shrd_wcnt;
    struct cv       sx_excl_cv;
    int             sx_excl_wcnt;
    struct proc     *sx_xholder;
};
```

They should be only used where the vast majority of accesses is shared.

- Create an *sx lock* with `sx_init()`.
- Attain a read (shared) lock with `sx_slock()` and release it with `sx_sunlock()`.
- Attain a write (exclusive) lock with `sx_xlock()` and release it with `sx_xunlock()`.
- Destroy an *sx lock* with `sx_destroy`.

Removing the Big Kernel Lock

These modifications made it possible to remove the Big Kernel Lock. The initial implementation replaced it with two mutexes:

- *Giant* is used in a similar manner to the BKL, but it is a blocking mutex. Currently it protects all entry to the kernel, including interrupt handlers. In order to be able to block, it must allow scheduling to continue.
- `sched_lock` is a spin lock which protects the scheduler queues.

This combination of locks supplied the bare minimum of locks necessary to build the new framework. In itself, it does not improve the performance of the system, since processes still block on *Giant*.

Idle processes

The planned light-weight interrupt threads need a process context in order to work. In the traditional UNIX kernel, there is not always a process context: the pointer `curproc` can be `NULL`. *SMPng* solves this problem by having an *idle process* which runs when no other process is active.

Recursive locking

Normally, if a lock is locked, it cannot be locked again. On occasions, however, it is possible that a process tries to acquire a lock which it already holds. Without special checks, this would cause a deadlock. Many implementations allow this so-called *recursive locking*. The locking code checks for the owner of the lock. If the owner is the current process, it increments a recursion counter. Releasing the lock decrements the recursion counter and only releases the lock when the count goes to zero.

There is much discussion both in the literature and in the FreeBSD SMP project as to whether recursive locking should be allowed at all. In gen-

eral, we have the feeling that recursive locks are evidence of untidy programming. Unfortunately, the code base was never designed for this kind of locking, and in particular library functions may attempt to reacquire locks already held. We have come to a compromise: in general, they are discouraged, and recursion must be specifically enabled for each mutex, thus avoiding recursion where it was not intended.

Migrating to fine-grained locking

Implementing the interrupt threads and replacing the Big Kernel Lock with Giant and sched-lock did not result in any performance improvements, but it provided a framework in which the transition to fine-grained locking could be performed. The next step was to choose a locking strategy and migrate individual portions of the kernel from under the protection of Giant.

One of the dangers of this approach is that locking conflicts might not be recognized until very late. In particular, the FreeBSD project has different people working on different kernel components, and it does not have a strong centralized architectural committee to determine locking strategy. As a result, we developed the following guidelines for locking:

- Use sleep mutexes. Spin mutexes should only be used in very special cases and only with the approval of the SMP project team. The only current exception to this rule is the scheduler lock, which by nature must be a spin lock.
- Do not `tsleep()` while holding a mutex other than Giant. The implementation of `tsleep()` and `cv_wait()` automatically releases Giant and gains it again on wakeup, but no other mutexes will be released.
- Do not `msleep()` or `cv_wait()` while holding a mutex other than Giant or the mutex passed as a parameter to `msleep()`. `msleep()` is a new function which combines the functionality with atomic release and regain of a specified mutex.
- Do not call a function that can grab Giant and then sleep unless no mutexes (other than possibly Giant) are held. This is a consequence of the previous rules.
- If calling `msleep()` or `cv_wait()` while holding Giant and another mutex, Giant must be acquired first and released last. This

avoids lock order reversals.

- Except for the Giant mutex used during the transition phase, mutexes protect data, not code.
- Do not `msleep()` or `cv_wait()` with a recursed mutex. Giant is a special case and is handled automagically behind the scenes, so don't pass Giant to these functions.
- Try to hold mutexes for as little time as possible.
- Try to avoid recursing on mutexes if at all possible. In general, if a mutex is recursively entered, the mutex is being held for too long, and a redesign is in order.

One of the weaknesses of the project structure is that there is no overall strategy for locking. In many cases, the choice of locking construct and granularity is left to the individual developer. In almost every case, locks are leaf node locks: very little code locks more than one lock at a time, and when it does, it is in a very tight context. This results in relatively reliable code, but it may not be result in optimum performance.

There are a number of reasons why we persist with this approach:

- FreeBSD is a volunteer project. Developers do what they think is best. They are unlikely to agree to an alternative implementation.
- We do not currently have enough architectural direction, nor enough experience with other SMP systems, to come up with an ideal locking strategy. This derives from the volunteer nature of the project, but note also that large UNIX vendors have found the choice of locking strategy to be a big problem.
- Unlike large companies, there is much less concern about throwaway implementations. If we find that the performance of a system component is suboptimal, we will discard it and start with a different implementation.

Migrating interrupt handlers

This new basic structure is now in place, and implementation of finer grained locking is proceeding. Giant will remain as a legacy locking mechanism for code which has not been converted to the new locking mechanism. For example, the main loop of the function `ithread_loop`, which

runs an interrupt handler, contains the following code:

```
if ((ih->ih_flags & IH_MPSAFE) == 0)
    mtx_lock(&Giant);
....
ih->ih_handler(ih->ih_argument);
if ((ih->ih_flags & IH_MPSAFE) == 0)
    mtx_unlock(&Giant);
```

The flag `INTR_MPSAFE` indicates that the interrupt handler has its own synchronization primitives.

A typical strategy planned for migrating device drivers involves the following steps:

- Add a mutex to the driver `softc`.
- Set the `INTR_MPSAFE` flag when registering the interrupt.
- Obtain the mutex in the same kind of situation where previously an `spl` was used. Unlike `spls`, however, the interrupt handlers must also obtain the mutex before accessing shared data structures.

Probably the most difficult part of the process will involve larger components of the system, such as the file system and the networking stack. We have the example of the BSD/OS code, but it's currently not clear that this is the best path to follow.

Kernel trace facility

The *ktr* package provides a method of tracing kernel events for debugging purposes. It is not intended for use during normal operation, and should not be confused with the kernel call trace facility *ktrace*.

For example, the function `sched_ithd`, which schedules the interrupt threads, contains the following code:

```
CTR3(KTR_INTR,
     "sched_ithd pid %d(%s) need=%d",
     ir->it_proc->p_pid,
     ir->it_proc->p_comm,
     ir->it_need);
...
if (ir->it_proc->p_stat == SWAIT) {
    CTR1(KTR_INTR,
        "sched_ithd: setrunqueue %d",
        ir->it_proc->p_pid);
```

The function `ithd_loop`, which runs the interrupt in process context, contains the following code at the beginning and end of the main loop:

```
for (;;) {
    CTR3(KTR_INTR,
        "ithd_loop pid %d(%s) need=%d",
        me->it_proc->p_pid,
        me->it_proc->p_comm,
        me->it_need);
    ...
    CTR1(KTR_INTR,
        "ithd_loop pid %d: done",
        me->it_proc->p_pid);
    mi_switch();
    CTR1(KTR_INTR,
        "ithd_loop pid %d: resumed",
        me->it_proc->p_pid);
```

The calls `CTR1` and `CTR3` are two macros which only compile any kind of code when the kernel is built with the `KTR` kernel option. If the kernel contains this option and the bit `KTR_INTR` is set in the variable `ktr_mask`, then these events will be masked to a circular buffer in the kernel. The *ddb* debugger has a command *show ktr* which dumps the buffer one page at a time, and *gdb* macros are also available. This gives a relatively useful means of tracing the interaction between processes:

```
2791 968643993:219224100
      cpu1 ../../i386/isa/ithread.c:214
      ithd_loop pid 21 ih=0xc235f200:
      0xc0324d98(0) flg=100
2790 968643993:219214043
      cpu1 ../../i386/isa/ithread.c:197
      ithd_loop pid 21(irq0: clk) need=1
2789 968643993:219205383
      cpu1 ../../i386/isa/ithread.c:243
      ithd_loop pid 21: resumed
2788 968643993:219190856
      cpu1 ../../i386/isa/ithread.c:158
      sched_ithd: setrunqueue 21
2787 968643993:219179402
      cpu1 ../../i386/isa/ithread.c:120
      sched_ithd pid 21(irq0: clk) need=0
```

The lines here are too wide for the paper, so they are shown wrapped as several lines. This example traces the arrival and processing of a clock interrupt on the i386 platform, in reverse chronological order. The number at the beginning of the line is the trace entry number.

- Entry 2787 shows the arrival of an interrupt at the beginning of `sched_ithd`. The second value on the trace line is the time since the epoch, followed by the CPU number and the file name and line number. The remaining values are supplied by the program to the `CTR3` function.
- Entry 2788 shows the second trace call in `sched_ithd`, where the interrupt handler is placed on the run queue.

- Entry 2789 shows the entry into the main loop of `ithd_loop`.
- Entries 2790 and 2791 show the exit from the main loop of `ithd_loop`.

Witness facility

The *witness* code was designed specifically to debug mutex code. It keeps track of the locks acquired and released by each thread. It also keeps track of the order in which locks are acquired with respect to each other. Each time a lock is acquired, witness uses these two lists to verify that a lock is not being acquired in the wrong order. If a lock order violation is detected, then a message is output to the kernel console detailing the locks involved and the locations in question. Witness can also be configured to drop into the kernel debugger when an order violation occurs.

The witness code also checks various other conditions such as verifying that one does not recurse on a non-recursive lock. For sleep locks, witness verifies that a new process would not be switched to when a lock is released or a lock is blocked on during an acquire while any spin locks are held. If any of these checks fail, the kernel will panic.

Project status

The project started in June 2000. The major milestones in the development are:

- June 2000: Ported the BSD/OS mutex code and replaced the Big Kernel Lock with `Giant` and `sched_lock`.
- September 2000: Replaced interrupt handlers with heavyweight interrupt processors. Initial commit to the FreeBSD source tree.
- November 2000: Made `softclock` MP-safe and migrate from under `Giant`.
- January 2001: Implemented condition variables.
- March 2001: Implemented read/write locks (called “shared/exclusive” or *sx* locks).
- March 2001: Complete locking of enough of the `proc` structure to allow signal handlers to be moved from under `Giant`.

The main issue in the immediate future is to migrate more and more code out from under `Giant`. In more detail, we have identified the fol-

lowing major tasks, some of which are in an advanced state of implementation:

- Split NFS into client and server.
- Add locking to NFS.
- Make the IP stack thread-safe.
- Create mechanism in `cdevsw` structure to protect thread-unsafe drivers.
- Complete locking struct `proc`.
- Cleanup the various `mp_machdep.c`’s, unify various SMP API’s such as IPI delivery, etc.
- Make `printf()` safe to call in almost any situation to avoid deadlocks.
- Make `mbuf` system use condition variables instead of `msleep()` and `wakeup()`.
- Remove the MP safe syscall flag from the system call table and add explicit `mtx_lock` of `Giant` to all system calls which need it.
- Use per-CPU buffers for *ktr* to reduce synchronization.
- Remove the priority argument from `msleep()` and `cv_wait()`.
- Implement lazy interrupt thread switching (context stealing).
- Lock structs `filedesc`, `pgrp`, `sigio`, `session` and `ifnet`.
- Make the virtual memory subsystem thread-safe.
- Convert `select()` to use condition variables.
- Reimplement *kqueue* using condition variables.
- Conditionalize atomic operations used for debugging statistics.
- Lock the virtual file system code.

Performance

The implementation has not progressed far enough to make any firm statements about performance, but we are expecting reasonable scalability to beyond 32 processor systems.

Acknowledgements

The FreeBSD SMPng project was made possible by BSDi's generous donation of code from the development version 5.0 of BSD/OS. The main contributors were:

- *John Baldwin* rewrote the low level interrupt code for i386 SMP, made much code machine independent, worked on the WITNESS code, converted `allproc` and `proctree` locks from `lockmgr` locks to `sx` locks, created a mechanism in `cdevsw` structure to protect thread-unsafe drivers, locked struct `proc` and unified various SMP API's such as IPI delivery.
- *Jake Burkholder* ported the BSD/OS locking primitives for i386, implemented `msleep()`, condition variables and kernel preemption.
- *Matt Dillon* converted the Big Kernel spinlock to the blocking Giant lock and added the scheduler lock and per-CPU idle processes.
- *Jason Evans* made `malloc` and friends thread-safe, converted simplelocks to mutexes and implemented `sx` (shared/exclusive) locks.
- *Greg Lehey* implemented the heavy-weight interrupt threads, rewrote the low level interrupt code for i386 UP, removed `spl`s and ported the BSD/OS `ktr` code.
- *Bosko Milekic* made `sf_bufs` thread-safe, cleaned up the mutex API and made the `mbuf` system use condition variables instead of `msleep()`.
- *Doug Rabson* ported the BSD/OS locking primitives, implemented the heavy-weight interrupt threads and rewrote the low level interrupt code for the Alpha architecture.

Further contributors were Tor Egge, Seth Kingsley, Jonathan Lemon, Mark Murray, Chuck Paterson, Bill Paul, Alfred Perlstein, Dag-Erling Smørgrav and Peter Wemm.

Bibliography

Per Brinch Hansen, *Operating System Principles*. Prentice-Hall, 1973.

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, *The Design and*

Implementation of the 4.4BSD Operating System, Addison-Wesley 1996.

Curt Schimmel, *UNIX Systems for Modern Architectures*, Addison-Wesley 1994.

Uresh Vahalia, *UNIX Internals*. Prentice-Hall, 1996.

Further reference

See the FreeBSD SMP home page at <http://www.FreeBSD.org/smg/>.

Page replacement in Linux 2.4 memory management

Rik van Riel
Conectiva Inc.

riel@conectiva.com.br, <http://www.surriel.com/>

Abstract

While the virtual memory management in Linux 2.2 has decent performance for many workloads, it suffers from a number of problems. The first part of this paper contains a description of how the Linux 2.2 VMM works and an analysis of why it has bad behaviour in some situations.

The way in which a lot of this behaviour has been fixed in the Linux 2.4 kernel is described in the second part of the paper. Due to Linux 2.4 being in a code freeze period while these improvements were implemented, only known-good solutions have been integrated. A lot of the ideas used are derived from principles used in other operating systems, mostly because we have certainty that they work and a good understanding of why, making them suitable for integration into the Linux codebase during a code freeze.

1 Linux 2.2 memory management

The memory management in the Linux 2.2 kernel seems to be focussed on simplicity and low overhead. While this works pretty well in practice for most systems, it has some weak points left and simply falls apart under some scenarios.

Memory in Linux is unified, that is all the physical memory is on the same free list and can be allocated to any of the following memory pools on demand. Most of these pools can grow and shrink on demand. Typically most of a system's memory will be allocated to the data pages of processes and the page and buffer caches.

- The slab cache: this is the kernel's dynamically allocated heap storage. This memory is

unswappable, but once all objects within one (usually page-sized) area are unused, that area can be reclaimed.

- The page cache: this cache is used to cache file data for both `mmap()` and `read()` and is indexed by (inode, index) pairs. No dirty data exists in this cache; whenever a program writes to a page, the dirty data is copied to the buffer cache, from where the data is written back to disk.
- The buffer cache: this cache is indexed by (block device, block number) tuples and is used to cache raw disk devices, inodes, directories and other filesystem metadata. It is also used to perform disk IO on behalf of the page cache and the other caches. For disk reads the pagecache bypasses this cache and for network filesystems it isn't used at all.
- The inode cache: this cache resides in the slab cache and contains information about cached files in the system. Linux 2.2 cannot shrink this cache, but because of its limited size it does need to reclaim individual entries.
- The dentry cache: this cache contains directory and name information in a filesystem-independent way and is used to lookup files and directories. This cache is dynamically grown and shrunk on demand.
- SYSV shared memory: the memory pool containing the SYSV shared memory segments is managed pretty much like the page cache, but has its own infrastructure for doing things.
- Process mapped virtual memory: this memory is administrated in the process page tables. Processes can have page cache or SYSV shared memory segments mapped, in which case those pages are managed in both the page tables and the data structures used for respectively the page cache or the shared memory code.

1.1 Linux 2.2 page replacement

The page replacement of Linux 2.2 works as follows. When free memory drops below a certain threshold, the pageout daemon (kswapd) is woken up. The pageout daemon should usually be able to keep enough free memory, but if it isn't, user programs will end up calling the pageout code itself.

The main pageout loop is in the function `try_to_free_pages`, which starts by freeing unused slabs from the kernel memory pool. After that, it calls the following functions in a loop, asking each of them to scan a small part of their part of memory until enough memory has been freed.

- `shrink_mmap` is a classical clock algorithm, which loops over all physical pages, clearing referenced bits, queueing old dirty pages for IO and freeing old clean pages. The main disadvantage it has compared to a clock algorithm, however, is that it isn't able to free pages which are in use by a program or a shared memory segment. Those pages need to be unmapped by `swap_out` first.
- `shm_swap` scans the SYSV shared memory segments, swapping out those pages that haven't been referenced recently and which aren't mapped into any process.
- `swap_out` scans the virtual memory of all processes in the system, unmapping pages which haven't been referenced recently, starting swapout IO and placing those pages in the page cache.
- `shrink_dcache_memory` reclaims entries from the VFS name cache. This is not directly reusable memory, but as soon as a whole page of these entries gets unused we can reclaim that page.

Some balancing between these memory freeing function is achieved by calling them in a loop, starting of by asking each of these functions to scan a little bit of their memory, as each of these functions accepts a priority argument which tells them how big a percentage of their memory to scan. If not enough memory is freed in the first loop, the priority is increased and the functions are called again. The idea behind this scheme is that when one memory pool is heavily used, it will not give up its resources lightly

and we'll automatically fall through to one of the other memory pools. However, this scheme relies on each of the memory pools to react in a similar way to the priority argument under different load conditions. This doesn't work out in practice because the memory pools just have fundamentally different properties to begin with.

1.2 Problems with the Linux 2.2 page replacement

- Balancing between evicting pages from the file cache, evicting unused process pages and evicting pages from shm segments. If memory pressure is "just right" `shrink_mmap` is always successful in freeing cache pages and a process which has been idle for a day is still in memory. This can even happen on a system with a fairly busy filesystem cache, but only with the right phase of moon.
- Simple NRU[Note] replacement cannot accurately identify the working set versus incidentally accessed pages and can lead to extra page faults. This doesn't hurt noticeably for most workloads, but it makes a big difference in some workloads and can be fixed easily, mostly since the LFU replacement used in older Linux kernels is known to work.
- Due to the simple clock algorithm in `shrink_mmap`, sometimes clean, accessed pages can get evicted before dirty, old pages. With a relatively small file cache that mostly consists of dirty data, eg unpacking a tarball, it is possible for the dirty pages to evict the (clean) metadata buffers that are needed to write the dirty data to disk. A few other corner cases with amusing variations on this theme are bound to exist.
- The system reacts badly to variable VM load or to load spikes after a period of no VM activity. Since `kswapd`, the pageout daemon, only scans when the system is low on memory, the system can end up in a state where some pages have referenced bits from the last 5 seconds, while other pages have referenced bits from 20 minutes ago. This means that on a load spike the system has no clue which are the right pages to evict from memory, this can lead to a swapping storm, where the wrong pages are evicted and almost immediately af-

terwards faulted back in, leading to the page-out of another random page, etc...

- Under very heavy loads, NRU replacement of pages simply doesn't cut it. More careful and better balanced pageout eviction and flushing is called for. With the fragility of the Linux 2.2 pageout framework this goal doesn't really seem achievable.

The facts that `shrink_mmap` is a simple clock algorithm and relies on other functions to make process-mapped pages freeable makes it fairly unpredictable. Add to that the balancing loop in `try_to_free_pages` and you get a VM subsystem which is extremely sensitive to minute changes in the code and a fragile beast at its best when it comes to maintenance or (shudder) tweaking.

2 Changes in Linux 2.4

For Linux 2.4 a substantial development effort has gone into things like making the VM subsystem fully fine-grained for SMP systems and supporting machines with more than 1GB of RAM. Changes to the pageout code were done only in the last phase of development and are, because of that, somewhat conservative in nature and only employ known-good methods to deal with the problems that happened in the page replacement of the Linux 2.2 kernel. Before we get to the page replacement changes, however, first a short overview of the other changes in the 2.4 VM:

- More fine-grained SMP locking. The scalability of the VM subsystem has improved a lot for workloads where multiple CPUs are reading or writing the same file simultaneously; for example web or ftp server workloads. This has no real influence on the page replacement code.
- Unification of the buffer cache and the page cache. While in Linux 2.2 the page cache used the buffer cache to write back its data, needing an extra copy of the data and doubling memory requirements for some write loads, in Linux 2.4 dirty page cache pages are simply added in both the buffer and the page cache. The system does disk IO directly to

and from the page cache page. That the buffer cache is still maintained separately for filesystem metadata and the caching of raw block devices. Note that the cache was already unified for reads in Linux 2.2, Linux 2.4 just completes the unification.

- Support for systems with up to 64GB of RAM (on x86). The Linux kernel previously had all physical memory directly mapped in the kernel's virtual address space, which limited the amount of supported memory to slightly under 1GB. For Linux 2.4 the kernel also supports additional memory (so called "high memory" or `highmem`), which can not be used for kernel data structures but only for page cache and user process memory. To do IO on these pages they are temporarily mapped into kernel virtual memory and the data is copied to or from a bounce buffer in "low memory".

At the same time the memory zone for ISA DMA (0 - 16 MB physical address range) has also been split out into a separate page zone. This means larger x86 systems end up with 3 memory zones, which all need their free memory balanced so we can continue allocating kernel data structures and ISA DMA buffers. The memory zones logic is generalised enough to also work for NUMA systems.

- The SYSV shared memory code has been removed and replaced with a simple memory filesystem which uses the page cache for all its functions. It supports both POSIX SHM and SYSV SHM semantics and can also be used as a swappable memory filesystem (`tmpfs`).

Since the changes to the page replacement code took place after all these changes and in the (one and a half year long) code freeze period of the Linux 2.4 kernel, the changes have been kept fairly conservative. On the other hand, we have tried to fix as many of the Linux 2.2 page replacement problems as possible. Here is a short overview of the page replacement changes: they'll be described in more detail below.

- Page aging, which was present in the Linux 1.2 and 2.0 kernels and in FreeBSD has been reintroduced into the VM. However, a few small changes have been made to avoid some artifacts of virtual page based aging.

- To avoid the eviction of "wrong" pages due to interactions from page aging and page flushing, the page aging and flushing has been separated. There are active and inactive page lists.
- Page flushing has been optimised to avoid too much interference by writeout IO on the more time-critical disk read IO.
- Controlled background page aging during periods of little or no VM activity in order to keep the system in a state where it can easily deal with load spikes.
- Streaming IO is detected; we do early eviction on the pages that have already been used and reward the IO stream with more aggressive readahead.

3 Linux 2.4 page replacement changes in detail

The development of the page replacement changes in Linux 2.4 has been influenced by two main factors. Firstly the bad behaviours of Linux 2.2 page replacement had to be fixed, using only known-good strategies because the development of Linux 2.4 had already entered the "code freeze" state. Secondly the page replacement had to be more predictable and easier to understand than Linux 2.2 because tuning the page replacement in Linux 2.2 was deserving of the proverbial label "subtle and quick to upset". This means that only VM ideas that are well understood and have little interactions with the rest of the system were integrated. Lots of ideas were taken from other freely available operating systems and literature.

3.1 Page aging

Page aging was the first easy step in making the bad border-case behaviour from Linux 2.2 go away, it works reasonably well in Linux 1.2, Linux 2.0 and FreeBSD. Page aging allows us to make a much finer distinction between pages we want to keep in memory and pages we want to swap out than the NRU aging in Linux 2.2.

Page aging in these OSes works as follows: for each physical page we keep a counter (called age in Linux,

or act_count in FreeBSD) that indicates how desirable it is to keep this page in memory. When scanning through memory for pages to evict, we increase the page age (adding a constant) whenever we find that the page was accessed and we decrease the page age (subtracting a constant) whenever we find that the page wasn't accessed. When the page age (or act_count) reaches zero, the page is a candidate for eviction.

However, in some situations the LFU[Note] page aging of Linux 2.0 is known to have too much CPU overhead and adjust to changes in system load too slowly. Furthermore, research[Smaragdis, Kaplan, Wilson] has shown that recency of access is a more important criteria for page replacement than frequency.

These two problems are solved by doing exponential decline of the page age (divide by two instead of subtracting a constant) whenever we find a page that wasn't accessed, resulting in page replacement which is closer to LRU[Note] than LFU. This reduces the CPU overhead of page aging drastically in some cases; however, no noticeable change in swap behaviour has been observed.

Another artifact comes from the virtual address scanning. In Linux 1.2 and 2.0 the system reduces the page age of a page whenever it sees that the page hasn't been accessed from the page table which it is currently scanning, completely ignoring the fact that the page could have been accessed from other page tables. This can put a severe penalty on heavily shared pages, for example the C library.

This problem is fixed by simply not doing "downwards" aging from the virtual page scans, but only from the physical-page based scanning of the active list. If we encounter pages which are not referenced, present in the page tables but not on the active list, we simply follow the swapout path to add this page to the swap cache and the active list so we'll be able to lower the page age of this page and swap it out as soon as the page age reaches zero.

3.2 Multiple page lists

The bad interactions between page aging and page flushing, where referenced clean pages were freed before old dirty pages, is fixed by keeping the pages which are candidates for eviction separated from the

pages we want to keep in memory (page age zero vs. nonzero). We separate the pages out by putting them on various page lists and having separate algorithms deal with each list.

Pages which are not (yet) candidate for eviction are in process page tables, on the active list or both. Page aging as described above happens on these pages, with the function `refill_inactive()` balancing between scanning the page tables and scanning the active list.

When the page age on a page reaches zero, due to a combination of pageout scanning and the page not being actively used, the page is moved to the `inactive_dirty` list. Pages on this list are not mapped in the page tables of any process and are, or can become, reclaimable. Pages on this list are handled by the function `page_launder()`, which flushes the dirty pages to disk and moves the clean pages to the `inactive_clean` list.

Unlike the active and `inactive_dirty` lists, the `inactive_clean` list isn't global but per memory zone. The pages on these lists can be immediately reused by the page allocation code and count as free pages. These pages can also still be faulted back into where it came from, since the data is still there. In BSD this would be called the "cache" queue.

3.3 Dynamically sized inactive list

Since we do page aging to select which pages to evict, having a very large statically sized inactive list (like FreeBSD has) doesn't seem to make much sense. In fact, it would cancel out some of the effects of doing the page aging in the first place: why spend much effort selecting which pages to evict[Dillon] when you keep as much as 33% of your swappable pages on the inactive list? Why do careful page aging when 33% of your pages end up as candidates for eviction at the same priority and you've effectively undone the aging for those 33% of pages which are candidates for eviction?

On the other hand, having lots of inactive pages to choose from when doing page eviction means you have more chances of avoiding writeout IO or doing better IO clustering. It also gives you more of a "buffer" to deal with allocations due to page faults, etc.

Both a large and a small target size for the inactive page list have their benefits. In Linux 2.4 we have chosen for a middle ground by letting the system dynamically vary the size of the inactive list depending on VM activity, with an artificial upper limit to make sure the system always preserves some aging information.

Linux 2.4 keeps a floating average of the amount of pages evicted per second and sets the target for the inactive list and the free list combined to the free target plus this average number of page steals per second. Not only does this second give us enough time to do all kinds of page flushing optimisations, it also is small enough to keep page age distribution within the system intact, allowing us to make good choices on which pages to evict and which pages to keep.

3.4 Optimised page flushing

Writing out pages from the `inactive_dirty` list as we encounter them can cause a system to totally destroy read performance because of the extra disk seeks done. A better solution is to delay writeout of dirty pages and let these dirty pages accumulate until we can do better IO clustering so that these pages can be written out to disk with less disk seeks and less interference with read performance.

Due to the development of the page replacement changes happening in the code freeze, the system currently has a rather simple implementation of what's present in FreeBSD 4.2. As long as there are enough clean inactive pages around, we keep moving those to the `inactive_clean` list and never bother with syncing out the dirty pages. Note that this catches both clean pages and pages which have been written to disk by the update daemon (which commits filesystem data to disk periodically).

This means that under loads where data is seldom written we can avoid writing out dirty inactive pages most of the time, giving us much better latencies in freeing pages and letting streaming reads continue without the disk head moving away to write out data all the time. Only under loads where lots of pages are being dirtied quickly does the system suffer a bit from syncing out dirty data irregularly.

Another alternative would have been the strategy used in FreeBSD 4.3, where dirty pages get to stay

in the inactive list longer than clean pages but are synced out before the clean pages are exhausted. This strategy gives more consistent pageout IO in FreeBSD during heavy write loads. However, a big factor causing the irregularities in pageout writes using the simpler strategy above may well be caused because of the huge inactive list target in FreeBSD (33). It is not at all clear what this more complicated strategy would do when used on the dynamically sized inactive list on Linux 2.4, because of this Linux 2.4 uses the better understood strategy of evicting clean inactive pages first and only after those are gone start syncing the dirty ones.

3.5 Background page aging

On many systems the normal operating mode is that after a period of relative activity a sudden load spike comes in and the system has to deal with that as gracefully as possible. Linux 2.2 has the problem that, with the lack of an inactive page list, it is not clear at all which pages should be evicted when a sudden demand for memory kicks in.

Linux 2.4 is better in this respect, with the reclaim candidates neatly separated out on the inactive list. However, the inactive list could have any random size the moment VM pressure drops off. We'd like get the system in a more predictable state while the VM pressure is low. In order to achieve this, Linux 2.4 does background scanning of the pages, trying to get a sane amount of pages on the inactive list, but without scanning aggressively so only truly idle pages will end up on the inactive list and the scanning overhead stays small.

3.6 Drop behind

Streaming IO doesn't just have readahead, but also its natural complement: drop behind. After the program doing the streaming IO is done with a page, we depress its priority heavily so it will be a prime candidate for eviction. Not only does this protect the working set of running processes from being quickly evicted by streaming IO, but it also prevents the streaming IO from competing with the pageouts and pageins of the other running processes, which reduces the number of disk seeks and allows the streaming IO to proceed at a faster speed. Currently readahead and drop-behind only work for

read() and write(); mmap()ed files and swap-backed anonymous memory aren't supported yet.

4 Conclusions

Since the Linux 2.4 kernel's VM subsystem is still being tuned heavily, it is too early to come with conclusive figures on performance. However, initial results seem to indicate that Linux 2.4 generally has better performance than Linux 2.2 on the same hardware.

Reports from users indicate that performance on typical desktop machines has improved a lot, even though the tuning of the new VM has only just begun. Throughput figures for server machines seem to be better too, but that could also be attributed to the fact that the unification of the page cache and the buffer cache is complete.

One big difference between the VM in Linux 2.4 and the VM in Linux 2.2 is that the new VM is far less sensitive to subtle changes. While in Linux 2.2 a subtle change in the page flushing logic could upset page replacement, in Linux 2.4 it is possible to tweak the various aspects of the VM with predictable results and little to no side-effects in the rest of the VM.

The solid performance and relative insensitivity to subtle changes in the environment can be taken as a sign that the Linux 2.4 VM is not just a set of simple fixes for the problems experienced in Linux 2.2, but also a good base for future development.

5 Remaining issues

The Linux 2.4 VM mainly contains easy to implement and obvious to verify solutions for some of the known problems Linux 2.2 suffers from. A number of issues are either too subtle to implement during the code freeze or will have too much impact on the code. The complete list of TODO items can be found on the Linux-MM page[Linux-MM]; here are the most important ones:

- Low memory deadlock prevention: with the arrival of journaling and delayed-allocation

filesystems it is possible that the system will need to allocate memory in order to free memory; more precisely, to write out data so memory can become freeable. To remove the possibility for deadlock, we need to limit the number of outstanding transactions to a safe number, possibly letting each of the page flushing functions indicate how much memory it may need and doing bookkeeping of these values. Note that the same problem occurs with swap over network.

- Load control: no matter how good we can get the page replacement code, there will always be a point where the system ends up thrashing to death. Implementing a simple load control system, where processes get suspended in round-robin fashion when the paging load gets too high, can keep the system alive under heavy overload and allow the system to get enough work done to bring itself back to a sane state.
- RSS limits and guarantees: in some situations it is desirable to control the amount of physical memory a process can consume (the resident set size, or RSS). With the virtual address based page scanning of Linux' VM subsystem it is trivial to implement RSS ulimits and minimal RSS guarantees. Both help to protect processes under heavy load and allow the system administrator to better control the use of memory resources.
- VM balancing: in Linux 2.4, the balancing between the eviction of cache pages, swap-backed anonymous memory and the inode and dentry caches is essentially the same as in Linux 2.2. While this seems to work well for most cases there are some possible scenarios where a few of the caches push the other users out of memory, leading to suboptimal system performance. It may be worthwhile to look into improving the balancing algorithm to achieve better performance in "non-standard" situations.
- Unified readahead: currently readahead and drop-behind only works for read() and write(). Ideally they should work for mmap()ed files and anonymous memory too. Having the same set of algorithms for both read()/write(), mmap() and swap-backed anonymous memory will simplify the code and make performance improvements in the readahead and

drop-behind code immediately available to all of the system.

6 Acknowledgements

The author would like to thank, in no particular order: Stephen Tweedie, for taking care of memory management in Linux 1.2, 2.0 and 2.2 and also for his help with this paper; Matt Dillon, for taking the time to explain the rationale behind every little piece of the FreeBSD VM; Conectiva Inc, who employ the author to hack the Linux kernel and the wonderful crowd testers from #kernelnewbies[Kernelnewbies] and elsewhere who have helped flesh out the bugs in the Linux 2.4 VM.

References

- [de Castro] Rodrigo S. de Castro
Linux 2.4 Virtual Memory Overview (2001)
<http://linuxcompressed.sourceforge.net/vm24/>
- [Dillon] Matthew Dillon
Design Elements of the FreeBSD VM System (2000)
http://www.daemonnews.org/200001/freebsd_vm.html
- [Kernelnewbies] Kernelnewbies
<http://kernelnewbies.org/>
- [Linux-MM] The Linux Memory Management home page
<http://linux-mm.org/>
- [Smaragdis, Kaplan, Wilson] Yannis Smaragdakis, Scott F. Kaplan and Paul R. Wilson
EELRU: Simple and Effective Adaptive Page Replacement, SIGMETRICS '99
<http://www.cs.amherst.edu/~sfkaplan/papers/index.html>
- [Note] Extensive documentation about page replacement algorithms is available practically everywhere. The 3 algorithms discussed in this paper are:
 - NRU: Not Recently Used, we scan through memory and evict every page that wasn't accessed since we last scanned it.

- LRU: we evict those pages that haven't been accessed for the longest time.
- LFU: we evict those pages that have been accessed least frequently in recent times.

User-Level Extensibility in the Mona File System*

Paul W. Schermerhorn Robert J. Minerick

Peter W. Rijks Vincent W. Freeh

Department of Computer Science and Engineering

University of Notre Dame

Notre Dame, IN 46556

{pscherml,rminerick,prijks,vin}@cse.nd.edu

Abstract

An extensible file system raises the level of file abstraction which provides benefits to both the end-user and programmer. The Modify-on-Access file system provides safe and simple user-defined extensibility through *transformations*, which are modular operations on input and output streams. A user inserts transformations into input and output streams, which modify the data accessed. Untrusted transformations execute in user space for safety. Performance of user-level transformations, although much slower than that of in-kernel transformations, is comparable to other user-level approaches, such as pipes.

This paper presents several interesting user-level transformations. For example, the `command` transformation executes a shell script whose input and output are routed from/to the file system. A file guarded by the `ftp` transformation is a “mount” point to an FTP server. The `php` transformation creates dynamic documents from PHP source when read. A file written to a sound device that is guarded by the `mp3` transformation is decoded on the fly, in the file system, before reaching the sound device.

Mona is a novel approach to file system extensibility that provides heretofore unseen flexibility. Mona is fine-grained: a user defines actions on a per-file basis. It is modular: transformations can be stacked upon one another. Mona supports two classes of transformations: kernel-resident and user-level.

1 Introduction

Unix-like operating systems have commonly used the file system to provide hardware abstractions for applications programmers. For example, Linux provides the devices

`/dev/audio` and `/dev/dsp` (among others) to provide easy access to sound devices. Placing these abstractions in the file system allows programmers to simply write to a file rather than have to go through the difficult process of passing data to a kernel module. This is one of the tasks of an operating system—to facilitate commonly-performed operations. However, the semantic function of the file system has changed little over the years. File systems can do more than provide generic access to hardware devices. Raising the semantic level of the file system provides benefits to both the end-user and the programmer.

Performing common tasks at the file system level allows applications to be written at a higher level. For example, a file system that can decode an MP3 audio file is able to provide a *decoder* file to which an application writes a raw MP3 file, rather than decoding and writing it to a device file. This relieves application programmers of the responsibility to implement widely shared functionality. Furthermore, portability will be enhanced if the programmer does not have to rewrite these common operations for every target platform. End-users benefit by having more stable software (because upgrades to common operations can be achieved more easily at a single common point) and greater functionality (because applications will leverage common operations more readily). The Modify-on-Access (Mona) file system provides safe and simple extensibility, allowing file system extensions to provide much of the functionality common to many applications.

Mona is an extensible file system based on Linux's `ext2` file system [9]. Mona has recently been ported to the 2.4 series of the kernel from the 2.2 series. Mona allows users to associate actions, called *transformations*, with the input and output streams of files. These transformations operate on the data before it is passed on to the user process accessing the file. This technique of pushing operations out of the application and into the file system extends the capabilities of traditional file systems. Furthermore, transformations may be stacked upon one

*This research was supported by NSF CAREER grant CCR-9876073, the JPL HTMT Project, the Arthur J. Schmitt Fellowship, and the ND Faculty Research Program.

another to create complex functionality out of simpler components.

The Mona file system supports both kernel and user-level extensions. Many existing schemes to extend file system functionality focus solely on kernel extensions. While there are many cases in which kernel extensions are the most appropriate mechanism, for many operations user-level extensibility is the better choice. We enumerate several reasons below.

- First, programming in user-space is much easier than in the kernel. Commonly available libraries (e.g., *libc*) can be used just as they can in application programs, whereas few functions are available to the kernel programmer.
- Second, debugging is much easier in user space than in the kernel.
- Third, time-consuming extensions are scheduled automatically by the system when implemented in user-space. On the other hand, because the Linux kernel is not preemptive, a long running extension implemented in the kernel must explicitly schedule itself.
- Finally, while it is obvious that no system can be completely safe from malicious users and faulty code, executing in user-space is much safer than executing in the kernel.

This paper describes user-level extensibility in the Mona file system [9, 10]. Untrusted user-defined transformations execute safely outside the kernel, modifying data as it is read and written. Users associate zero or more transformations with input and output streams of a file, which specialize the file system for a particular application or use. Although executing transformations in user-space is less efficient than executing in the kernel, user-level transformations are efficient. Read and write operations are approximately five times slower when guarded by user-level transformations than when guarded by kernel transformations. This overhead is not noticeable for interactive operations, and is comparable to the performance of Unix pipes.

Additionally, this paper describes several interesting user-level file transformations, illustrating the possible uses of a higher-level file system. For example, the *ftp* transformation provides transparent access to files on a remote FTP server. On a file access, the *ftp* transformation issues an FTP request to the designated remote site, waits, and fills local buffers with the returned data. The *php* transformation reads a raw PHP file from disk. It parses the contents of the file and creates data for the file buffers dynamically. Further, the *mp3* transformation decodes MP3 files on the fly. Thus, an application

can simply write a raw MP3 file to a special file. The file system takes care of decoding. Lastly, the *command* transformation executes a program—often a simple shell script. The *command* transformation redirects the I/O of the program from/to the file system. As a result, extending the Mona file system is as simple as writing a shell script.

The remainder of this paper is organized as follows. Section 2 provides an overview of the Mona file system. Section 3 describes several transformations. Section 4 discusses the *export* transformation that safely executes user-level transformations in user space. Section 5 presents measurements of the Mona file system and the *export* transformation. Section 6 discusses related work. The last section presents our conclusions.

2 Mona File System Overview

Mona provides file system extensions as *transformations*, which modify streaming data [9]. A typical transformation acts as a filter, reading input, then pushing modified data downstream. Mona supports two types of transformations: *kernel* and *user-level*. Common kernel transformation code is downloaded into the kernel at the time the Mona kernel module is inserted. Less frequently used kernel transformations may also be added (and removed) dynamically. User-level transformations are executed in a user-level process that communicates with the file system via the *export* kernel transformation, as described in Section 4.

The Mona file system creates *virtual* files, whose contents exist only in the file system while the virtual file is open. There are two mechanisms for creating virtual files: persistent and transient. A *persistent* transformation link, which is similar to a symbolic link, creates an access point in the file system. Any program can access a transformation link exactly as it would access an ordinary file. A persistent transformation link exists until it is explicitly removed.

The *lnx* utility creates persistent transformation links—it is similar to *ln*. In the example below, the *lnx* utility is used to create a persistent link to the file */dev/audio* with the *fail* transformation on the input (read) and the *mp3* transformation on the output (write).

```
lnx /dev/audio mp3 -r fail_xform \  
-w mp3_xform
```

When a program writes to the virtual file, *mp3*, the MP3 data that the program writes is decoded and written directly to the audio device */dev/audio*. In this case, there is no reason to read the virtual file, and so the *fail* transformation guards the input stream. In this persistent

case, the virtual file exists for all programs to use until the user explicitly deletes it.

Conversely, a *transient* transformation exists only as long as the virtual file is open. A user creates a transient data view by manipulating transformations at runtime. A user pushes and pops transformations on the data streams of an open file using the `ioctl` system call. The example below pushes the `fail` transformation on the input stream and the `mp3` transformation on the output stream.

```
ioctl(fd, PUSH_INPUT,
      "fail_xform /dev/audio");
ioctl(fd, PUSH_OUTPUT,
      "mp3_xform /dev/audio");
```

This creates a virtual file identical to the persistent transformation example above. However, the virtual file is only accessible through the file descriptor, and is destroyed when the file is closed.

Mona adds new functionality to the file system without sacrificing backwards compatibility. Because Mona is compatible with `ext2`, either file system may be used to read and write media based on the other (of course, `ext2` does not support the added functionality of Mona). The Mona file system is a Linux kernel module, which can be loaded and unloaded as needed.

In addition to maintaining compatibility with the `ext2` file system, we have also demonstrated that the overhead of using the Mona file system is negligible. When using Mona as a traditional file system, i.e. without utilizing its enhanced capabilities, an overhead of less than 1% is incurred on `read`, `write`, and `open` system calls. Additionally, tests have shown that Mona performs similarly to `ext2` for the PostMark suite of benchmarks, the Andrew benchmark, and for kernel compilation. When utilizing user-level transformations we have found that little performance is sacrificed relative to what we believe is a large gain in functionality. In cases where a network of transformations is used, we have found that Mona can perform better than current models. Preliminary experiments have shown this to be true for emulating the functionality of Unix pipes, where we have shown a clear performance advantage [8].

3 User-level Transformations

This section first presents a number of user-level transformations that we have implemented to address real-world problems. These transformations allow existing mechanisms to be applied more broadly via the file system, allowing users to solve new problems or to solve existing problems more easily. After demonstrating some uses of user-level transformations, a brief tutorial on constructing user-level transformations is presented. Fi-

nally, the section concludes with a short discussion of the benefits of user-level file system extensions.

3.1 Applications of User Transformations

An active file system raises the level of the file system abstraction. The higher level of abstraction provides significant benefit to both developers and end-users.

- Application code development is simpler. First, there is less for the application to implement. Second, code re-use is more likely because programmers can use the well understood file system interface instead of having to learn a new API. Moreover, there is a central location for code, which simplifies upgrading systems.
- All applications can use the expanded capability—even those that are not modified. For example, encryption in the file system provides security to any application that reads or writes data from a file.
- Novel file system semantics and structures can be created. For example, locking or data consistency can be provided by the file that is concurrently shared—rather than by the applications (which may not even be aware the file is shared). Other examples are remote access (FTP, HTTP), logging, journaling, dynamic file creation (PHP), and file conversion.

The above are the potential advantages of an active file system. It is also a sampling of the potential uses of user-level transformations. Mona's flexibility and ease of programming allow for many other uses.

Mona user-level transformations give programmers a novel interface for code reuse. One example of this is the `mp3` transformation, which decodes MP3 audio files on the fly. Suppose a virtual file guards the audio device with the `mp3` transformation. Any program can now write an MP3 encoded file to the virtual file. The `mp3` transformation decodes the data in the file system, which is given to the device. Not only do programmers avoid the need to implement their own MP3 decoders, but having the code at a centralized point makes updates easier. Moreover, the code is associated with the device, not the applications.

Some user-level transformations provide new ways of using existing resources. The `ftp` transformation creates a common interface to local and remote files, allowing users to navigate a remote FTP site as if it were part of the local file system. A user executes ordinary Unix executables (e.g., `cd`, `ls`, `cp`, etc.) to manipulate remote files, eliminating the need for explicit file transfer requests. A virtual file named `foo.remote` can be a link to a file `foo` on a remote FTP site. When the user

reads the virtual file, the `ftp` transformation automatically negotiates with the remote host and transfers `foo` to the user's local machine. Except for the latency of the file transfer, the user may never know that the base file exists only on a remote server. For public FTP access, a user can use the generic `ftp` transformation. If a user wants to access private data via FTP, she can recompile the transformation to contain her username and password and place the new shared library in her personal transformation directory.

A variety of archive transformations allow access to the content of various archival formats, including `tar`, `rpm`, `deb`, `zoo`, and `zip` files without unpacking the archive. These transformations allow users read and write access to the various archives without the need to manually extract and re-archive those files. The similar `gzip` and `bzip2` transformations allow the same type of manipulation on files in compressed form. The `ftp`, `archive`, and `compression` user-level transformations all demonstrate Mona's flexibility and usefulness in providing new views and interfaces to existing resources.

User-level transformations can make use of existing tools. The `php` transformation is an example of this. PHP is a server-side, cross-platform, HTML embedded scripting language. It allows developers to create scripts that dynamically generate web pages. PHP is the most popular module for the Apache web server, and is employed in creating dynamic content for a large number of commercial web sites. The Mona `php` transformation combines the power of PHP with the flexibility of the Mona programming model. By moving PHP into a Mona user-level transformation, we allow the creation and delivery of dynamic content via any transport method (e.g. FTP, text editor, web browser, etc.).

Many web sites use PHP to parse a database file and generate the day's or week's events on the fly. Using the Mona PHP transformation, one can create a similar utility that does not depend on server processing. Consider guarding a user's `.plan` file with the `php` transformation. The raw data in the `.plan` is a PHP script that parses a database file. When any user reads this `.plan`, the `php` transformation dynamically generates a user's plan from a database file that is specific to the current date. This is an example of Mona's ability to extend the usefulness of an existing tool: e.g., serverless PHP.

The `command` transformation makes programming file system extensions even easier. Comparable in some ways to a traditional Unix command-line pipeline, this transformation allows the user to execute arbitrary executables or shell scripts on input and output streams. Text processing tools such as `grep`, `sed`, and `gawk` can be placed on streams to automatically filter out unwanted data. For example, if a user wants to print out only the first field of each line, a transformation link `bar.first`

can be created pointing to a file `bar.raw` with the below transformation on the input stream.

```
command /usr/bin/gawk '{print $1}'
```

The file `bar.first` appears as a normal file containing just the first field of the records in the file `bar.raw`.

There are many other potential uses of the `command` transformation. For example, a simple script could guard an HTML file, which forces a reload of a browser whenever the base HTML changes. In this way, users edit HTML files with their favorite editor, but can view up-to-date renderings of the files in web browsers. In another example, version control could be automated using the `rcs` utility. Finally, if a user wishes to be notified when a file is read or modified, the `command` transformation can send mail automatically. In short, any program can be associated with a data stream using the Mona file system. This illustrates the flexibility that the Mona system affords the user. Just as the `export` transformation is a kernel-resident transformation built to allow transformations to be executed in user space, the `command` transformation is a standard user-level transformation built to allow executables to be associated with data streams.

3.2 Implementing a User Transformation

This section describes the construction of simple user-level transformations. The implementation is primarily a matter of creating a filter that conforms to a well-defined function-call interface. A transformation has the following calling interface:

```
int decompress_xform(  
    XformInfoPtr xf_ptr,  
    int input_bytes,  
    char *buffer,  
    int *state);
```

The transformation info pointer, `xf_ptr`, uniquely identifies each instance of a transformation. In particular, it contains information about data in the transformation network and a pointer to memory that may be utilized by the user to store data private to the current instance of the transformation. The parameter `input_bytes` provides the size of the data passed to the transformation via the character pointer `buffer`. The last of the arguments passed to a transformation is `state`, an integer pointer to the "global" transformation state variable. Each transformation returns the total number of bytes placed in the buffer to be sent downstream to the next transformation in the network, if one exists. Additionally, it sets `state`, indicating whether all input has been consumed.

Figure 1 lists the source code of a simple decompression transformation. It takes an input buffer and passes it to a decompression routine. The decompression routine


```

int decompress(char *, char *, int);
typedef struct decomp_data { int size; int sent; char* data; } d_data;
int dec_xform(XformInfoPtr xf_ptr, int input_bytes, char *buffer, int *state) {
    d_data *data_ptr = (d_data *)xf_ptr->private_data;
    int output_bytes;

    if(decomp_data == NULL) {
        decomp_data = (d_data *)malloc(sizeof(d_data));
        decomp_data->size = decompress(buffer, data_ptr->data, input_bytes);
        decomp_data->sent = 0;
    }
    if((output_bytes = decomp_data->size - decomp_data->sent) > BLOCK_SIZE) {
        memcpy(buffer, decomp_data->data, BLOCK_SIZE);
        decomp_data->sent += BLOCK_SIZE;
        *state = XF_CURRENT_HAS_DATA;
    } else {
        memcpy(buffer, decomp_data->data, output_bytes);
        free(decomp_data->data);
        free(decomp_data);
        xf_ptr->private_data = NULL;
        *state = XF_READY;
    }
    (output_bytes > BLOCK_SIZE) ? return(BLOCK_SIZE) : return(output_bytes);
}

```

Figure 1: Source code for the *decompress* transformation.

is responsible for managing the space required for the new data, using the pointer passed by the transformation to allocate memory. If the `private_data` element of the structure `xf_ptr` is `NULL`, then a new block of data has been sent. In that case, some initialization tasks are performed, along with the decompression itself. If there is more data than can fit in one block, one block is sent along, and state is set to `XF_CURRENT_HAS_DATA`. This ensures that the current transformation will be called again before more data from the source file is passed in the buffer, allowing `decompress_xform` to continue sending the decompressed data until it has been consumed. Once the remaining decompressed data is smaller than the block size, it can be sent, cleanup can be performed, and state can be set to `XF_READY`, indicating that the transformation is ready for new data.

To make use of this transformation, the user must compile it as a shared library and put the shared library where Mona can find it:

```

gcc decompress_xform.c -shared \
    -o decompress.so
mv decompress.so ~/.mona

```

Mona user-level transformations are implemented as functions. When size is conserved, the programmer does not have to be concerned about setting flags or computing return values. More complex transformations require the programmer to maintain buffers, set state variables,

and compute return values. These additional programming tasks are minor in comparison to the added work of creating more complex transformations. When complete, the function is compiled into a shared library and is ready to use.

3.3 Advantages of User Transformations

The Mona API makes transformation development easy for the programmer. While kernel transformations in Mona share the same interface as user-level transformations, there are important advantages to programming user-level transformations. User-level transformations offer the programmer access to the libraries of code that have become a part of every programmer's tool box. Common debugging tools can also be used on user-level transformations. Additionally, since kernel transformations run in kernel mode, they are not preempted. This means that kernel transformations must be explicitly scheduled by the programmer. Because user-level transformations are scheduled like any other user-level process, they do not suffer from these complications. These three advantages of user-level transformations together with Mona's interface make user-level transformations an easy target for any developer.

In addition to the advantages discussed above, user-level transformations are also much safer than kernel transformations. Since user-level transformations run in

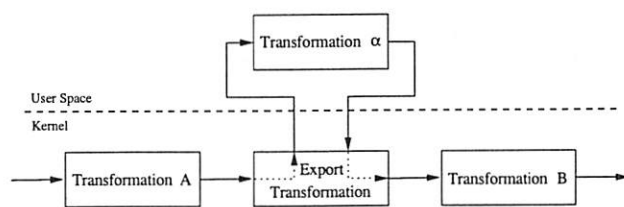


Figure 2: Illustration of export transformation.

user space, they cannot access internal kernel data structures and therefore cannot compromise the integrity of the kernel. This means that any user can write and test a user-level transformation without the intervention of a system administrator.

Mona user-level transformations provide users with a virtually unlimited number of ways to extend the file system. The ability to use existing code (and even existing applications) greatly reduces implementation time even for complex extensions. Mona's fine-grained per-file approach makes it more useful than many previous approaches to file system extension.

4 Export Transformation

In order to realize the advantages of executing in user space Mona needs a mechanism for exporting transformations. The export kernel transformation safely executes untrusted or computationally expensive transformations outside the kernel in user space. For example, consider executing three consecutive transformations, *A*, α , and *B*, where *A* and *B* are kernel-resident transformations and α is a user transformation. Mona inserts the export transformation between *A* and *B*, as shown in Figure 2. The export transformation passes its input data across the kernel/user space boundary to a user-level transformation α . When transformation α completes, it returns output data to the export transformation which then passes it on to *B*.

4.1 Export Execution Model

The export transformation executes user transformations in user-level processes for two reasons. First, processes allow the Mona implementation to easily support transformation concurrency. Second, the *setuid* system call provides control over the access permissions of a process (and any transformations within it).

The Mona implementation uses a daemon to supervise all transformations that execute outside of the kernel. When the file system instantiates an export transformation during an *open* system call, the daemon forks

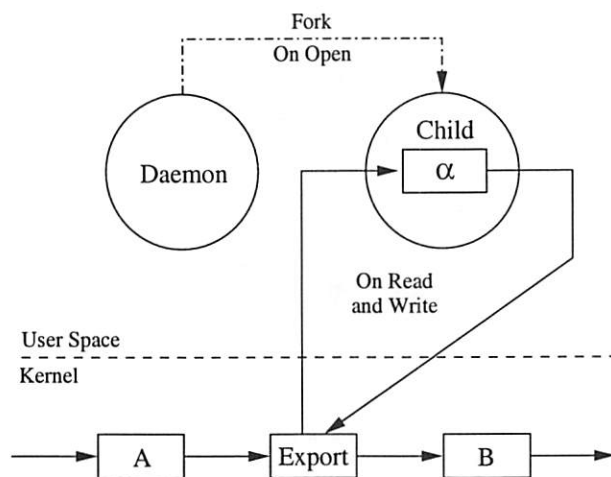


Figure 3: Execution model of export transformation.

a child process to handle accesses to the file. Any subsequent read or write to the file passes data up to the child process, which transforms the data and returns it to the export transformation, as shown in Figure 3. This figure provides an in-depth look at the example first introduced in Figure 2. Transformations *A* and *B* reside within the kernel, but α is exported to user space. Any data that streams through the network passes up through the export transformation to user space, through α , the transformation in the child process, and back to the kernel.

4.2 Export Implementation

The export transformation requires three extensions to the Mona file system. First, we extend the *ioctl* system call to enable communication between the kernel and user-space I/O streams. The `MONA_IOC_K2U_MASTER` option to the *ioctl* call allows the Mona daemon to detect when the file system instantiates a new export transformation. Two other *ioctl* options, `MONA_IOC_K2U_SLAVE` and `MONA_IOC_U2K`, allow children of the daemon to request and submit transformation data. Both request *ioctl* options block their calling processes until data is available. As a result, there is a clean, well-defined interface between the export transformation and the Mona daemon.

Second, Mona implements an initialization queue, which is a queue of transformations that are waiting to be pushed into user space. The Mona options for the *ioctl* system call give the Mona daemon access to this queue. The daemon blocks on the `MONA_IOC_K2U_MASTER` *ioctl* call until the file system places a transformation in the initialization queue. When this occurs the daemon awakens and performs the following

actions to initialize the transformation in the queue. The daemon reads a key from the kernel that uniquely identifies the transformation. Then it forks a child process and sets the UID and GID (user and group ID) of the child to that of the owner of the transformation, in order to preserve file permissions. (Section 4.3 discusses the security issues in detail.) The child process services subsequent requests, as described below.

The final extension required is an execution queue of initialized transformations that are awaiting execution. The file system moves a transformation from the initialization queue to the execution queue after the Mona daemon forks a child process for the transformation. The child process blocks in the queue until there is data on which it can execute. Included with the transformation data are the names of the shared library and the function to be called. Each child then opens the shared library file using the Linux `dlopen` call and loads the appropriate function with the `dlsym` call. Transformation data on the execution queue is uniquely identified by transformation identification keys. A child of the daemon uses its identification key, which was provided by the Mona daemon, as an argument to the `ioctl` `MONA_IOC_K2U_SLAVE` system call in order to request data from the execution queue. After the child transforms its data, it sends a reply back to the kernel through another `ioctl` call using the `MONA_IOC_U2K` option. This process repeats until the file access completes and the child terminates.

4.3 Export Security

For the exported transformation to be safe, it must execute with proper permissions and maintain the integrity of the kernel. Before a child of the Mona daemon executes a transformation, it changes its UID and GID to a safe permission level. There are three obvious choices for a UID and GID in this situation, the owner of the base file, the owner of the virtual file, or the user accessing the virtual file. However, two of these are unsafe. If a transformation ran under the permissions of the user accessing a file, the transformation creator would have access to the user's files. Setting a transformation's permissions to that of the owner of the base file is also unsafe. For example, a user could point a transformation link to a file owned by root and have the transformation generate a shell which has root permissions. Therefore, the Mona daemon uses the `setuid` system call to change the effective user id of the child process to that of the user who created the virtual file and specified the code to be executed [17]. Consequently, a transformation will not perform actions that the owner of the virtual file is not allowed to perform.

The `export` transformation does not compromise the integrity of the kernel even though data originating in

user space flows into the kernel. First, the mechanism for passing data across the kernel-user boundary truncates the kernel buffer if the child process, which is executing a user transformation, attempts to exceed the space allocated for the kernel buffer. Second, data that passes through a transformation is never executed, it is only appended to an I/O stream on a read or write.

Furthermore, the Mona file system enforces proper use of the `ioctl` extensions. The `MONA_IOC_K2U_MASTER` option restricts accesses to processes owned by root and exits with an error message for any other user. The other new options, `MONA_IOC_K2U_SLAVE` and `MONA_IOC_U2K`, allow any user to request and submit transformation data. However, each call requires a valid transformation identification key before the kernel accepts the call. It is conceivable that a user could randomly guess keys and attempt to insert or remove data from another user's I/O stream. However, with a large enough key the probability of successfully guessing a random key is essentially zero.

In summary, the `export` transformation overcomes the three difficulties of kernel-resident transformations. First, the Mona daemon allows an unprivileged user to extend file system capabilities in a secure manner. Second, transformation code that executes for extended periods of time runs in user space, where it is time shared along with all other processes to maintain fair scheduling of resources. However, when performance is paramount, one can implement a kernel-resident transformation. Finally, user-level transformations are much easier to implement than kernel-resident transformations due to the availability of user-space tools such as libraries and debuggers.

5 Results and Evaluation

Mona provides greater functionality through file system extensions. As such, the focus of the project has not been quantitative (i.e., user-level transformations do not focus on increasing system performance). Using Mona transformations will provide performance wins in some cases, but the purpose of Mona is to provide ways to make systems more useful to users. However, if the cost of a user-level transformation is large, its benefits could be outweighed. For this reason, we performed several tests to measure the overhead that Mona adds to a system.

To determine the baseline overhead associated with using Mona instead of `ext2`, we compare system call execution times for files in `ext2` and unguarded (i.e., no transformations) files in Mona. The results indicate that the overhead added to these system calls is less than 1%. This overhead applies only to the `open`, `read`, and `write` system calls. To find out the effect using Mona

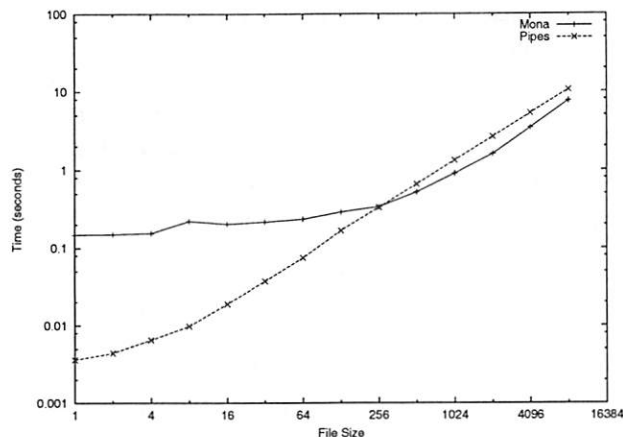


Figure 4: Unix pipes vs. Mona user-level transformations (Linux 2.2 kernel)

has on aggregate system performance, we also performed tests using several file system benchmark suites. Our tests have shown that the PostMark suite had the largest overhead at 2%, whereas the Andrew and kernel compile tests were both below 1%. This amount of overhead on unguarded files is small, and the added functionality provided by the Mona file system outweighs the performance penalty [8].

Mona's ability to allow unprivileged users to extend the file system is one of its key novel aspects. The overhead of using a Mona user transformation under the Linux 2.2 kernel is approximately 150 μ s. This cost is quickly amortized as file size increases, and is negligible for large files. The same principle applies for transformations of increasing complexity. For a transformation that is computationally non-trivial, the latency contributed by this overhead will be considered acceptable by the user.

In many cases the added functionality is similar to that of standard Unix pipes. To compare our performance to that of pipes, we ran several tests implementing the same computation as both processes communicating via pipes and as Mona transformations. This test sends input data through two transformations using a Unix pipe and the Mona file system. Figure 4 shows that Mona transformations compare favorably with Unix pipes. Due to the overhead of instantiating the new process, user transformation performance is worse than that of pipes until the file size approaches 128K, although only by a little over a tenth of a second in the worst case. For file sizes over 128K, Mona user-level transformations perform better than their pipe counterparts by as much as 65%. The two Mona user-level transformations operate in the same user process. Consequently there is no overhead from switching processes. Mona incurs no over-

head from buffer copying because it passes a pointer to a buffer between transformations. Mona's performance advantage increases when several operations are stacked upon one another.

6 Related Work

An adaptive I/O subsystem was implemented in Streams [16, 2, 13] and is a component of several System V variants. A stream is a connection between a device driver and a user process. The `ioctl` system call pushes stream modules (similar to Mona transformations) into the stream. When data flows through the stream and reaches a module, code from the module executes on the data before passing modified data downstream. Like Mona, the Streams system enables dynamic extensibility. However, there are some differences between the systems. First, all Stream modules execute with full permissions in the kernel. Consequently, only privileged and expert users can create new extensions. In addition, a Stream module is inherently duplex and adds a stage in both the input and output pipeline. This works well for operations that have natural inverses, such as compression/decompression. However, if an operation does not have or require an inverse operation (such as PHP) for data traveling the opposite direction, the technique wastes resources and adds an extra pipeline stage.

The watchdog system provides extended file semantics by guarding file accesses with special processes [3]. Each watchdog process is a user-level program associated with either a file or directory. When a guarded file opens, the kernel negotiates with the watchdog guarding the file to determine how to handle accesses. Like user-space transformations, watchdog processes provide a simple mechanism to add user-defined extensibility to a file system. However, creating a new process for each open guarded file is expensive in system resources and interprocess communication. Managing an entire process is excessive overhead for simple transformations, such as *lock*. The watchdog system cannot push common operations into the kernel where they can execute quickly. The flexibility of the Mona system allows the user to decide the best execution environment for any particular operation.

BSD Portals extend the file system by exporting certain open system calls to a user-space daemon [12]. A portal daemon is mounted as a standard file system. When the kernel resolves a pathname that leads through the portal daemon mount, the remainder of the path is sent to the daemon. Depending on the type of daemon that is mounted, some type of open occurs, and a file descriptor is returned. This allows for arbitrary code to be executed on opens, but this functionality is specific to the open system call. Translators provided

by GNU/Hurd [14] are very similar to Mona user-level transformations. A translator is a program inserted between the content of a file and the user process. Translators are user programs, and as such can be installed and modified by regular users. Translators provide the file system interface for Hurd programs.

Stackable file systems derive functionality from pre-existing file systems [6, 11]. By stacking a file system on top of another in a file system hierarchy, the operations provided by the lower level are inherited by the higher. A stackable file system is most effective when a handful of operations are required for a large set of files and the operations change infrequently. However, because a system administrator must implement all stacking, the benefit to ordinary users will be somewhat limited. Furthermore, the layering structure (and thus the functionality) of stackable file systems cannot be modified dynamically. Stackable file systems use mount points rather than files as targets for extensibility, and as such are a much more coarse-grained approach than Watchdogs or Mona, where users define their own operations and implement them on a per-file basis. Apollo's DOMAIN file system [15] is quite similar to stackable file systems. It allows users to define new file types and associate user defined procedures with these new types. However, extensible code resides in user processes.

Stackable templates alleviate the usability difficulties of stackable file systems by abstracting complex kernel code into templates [20]. Consequently, *Wrapfs*, a stackable template file system, provides a much simpler (and more usable) interface than previous stackable file systems by hiding details of the operating system internals. *Wrapfs* extends the vnode interface to enable stacking, as originally proposed by Rosenthal [18]. As a result, *Wrapfs* supports unmodified native file systems while providing users an extended vnode interface. This approach contrasts the Mona file system, which is implemented as a peer to other native file systems within an unmodified virtual file system interface, and maintains full compatibility with the *ext2* file system. Furthermore, unlike Mona, *Wrapfs* does not allow streams to change size at runtime.

A simulation of Active Disks uses transformation-like *disklets* to operate on data streams entering and exiting intelligent disk drives [1]. The Active Disk architecture integrates processors and large amounts of memory onto a disk drive. An analysis of several algorithms on this architecture found that an Active Disk using application-specific disklets outperforms conventional disk drives. Additionally, Active Disks scale considerably better than traditional disk architectures. Active Disks and Mona demonstrate the potential of modular, stream-oriented processing.

The *userfs* package implements customizable file sys-

tems as user processes [5]. Unlike the stackable systems discussed above, mounting a user file system does not require privileged access. As a result, unprivileged users can customize their environments without the assistance of a system administrator. This system thus allows simple file system extensibility, but is more coarse-grained than Mona. Entire hierarchies are mounted with the *userfs*, whereas the flexibility of the Mona file system allows actions to be associated with individual files or hierarchies.

There are many projects whose goal is to provide general kernel extensibility, including file system extensibility [4, 7, 19]. These systems address the safety issues associated with downloading untrusted code into the kernel. The Mona file system provides only a subset of the extensibility offered by general kernel extensibility projects (i.e., the subset that relates to the file system). Mona only allows trusted code in the kernel and concentrates on user-space extensibility for untrusted code.

Through the use of traditional Unix tools like filter programs and pipes, we can achieve a series of stacked operations on a data stream, but to use these, we need to either explicitly create the pipes in a process, or use the pipe functionalities of a shell. The former requires significant modification to applications, while the latter allows us to only operate on standard input or output, not arbitrary files. Shell associations perform actions (e.g., launching an application) based on the type of the underlying file. This could be similar to Mona's functionality in some cases, but does not, for example, allow dynamic insertion and deletion of functionality at runtime. Another Unix mechanism that can be used is the LD_PRELOAD functionality of dynamic libraries. Library preloading can be used to intercept file system calls and perform operations on the data between the user program and the operating system. However, when using library preloading, all calls that are intercepted will have to go through the filter code. Mona allows filters to be specified for files on an individual basis.

7 Conclusions

This paper describes how to raise the level of abstraction provided by a file system. It presents the Modify-on-Access (Mona) file system, which supports safe and simple user-defined extensions called transformations, which are modular, stream-oriented operations that are inserted into an I/O stream during a file access. This paper presents several examples in which extending the structure and semantics of a file simplifies applications and benefits both the application programmer and the end user.

The Mona file system provides a novel combination of granularity, modularity, and usability. Mona supports

transformations on a fine-grained per-file basis. In addition, transformations are modular and can compose larger operations. Finally, Mona can execute a transformation within the kernel or in a user-space process. As a result, a user has the ability to choose appropriate levels of performance, safety, and ease of use. The flexibility in all three areas distinguishes Mona from previous extensible file systems.

This paper presents several lessons. First, although transformations are a limited form of computation, there are many useful operations that fit the form. Moreover, it shows that the model (and our implementation) are ideally suited for extensions. The `export` transformation is a kernel transformation that enables user-level transformations. Similarly, the `command` transformation is a user-level transformation that enables shell script transformations.

This paper examines the cost of user-level transformations. We show that the cost is comparable to pipes, and in some cases better. The cost of user-level transformations is negligible when a transformation involves a latency-dependent operation, like the `ftp` transformation. Additionally, as the complexity of a transformation increases, the relative cost of a user-level transformation decreases. Even in the worst case—a trivial transformation—the absolute cost of invoking a user-level transformation is low enough (approximately 150 μ s) that there is not a noticeable additional delay for interactive environments.

Finally, this paper shows a simple but effective technique for maintaining security with user-level extensions. Our technique sets the UID and GID of the child helper processes such that code executes with permissions that are allowable by the system. Consequently, Mona ensures that the use of user-level transformations will never compromise security.

Acknowledgements

We would like to thank our reviewers for their comments, and especially Alan Nemeth for his help in shepherding us through the review and submission process.

We would also like to thank Richard Kendall for his foundational contributions to the Mona project.

Mona is available as open source in conformance with the Open Source Initiative's Open Source Definition. Mona is available at <http://www.cse.nd.edu/~ssr/projects/mona>.

References

- [1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proc. Eighth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.
- [2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.
- [3] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the UNIX file system. *USENIX Winter Conference*, pages 267–275, Winter 1988.
- [4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. Fifteenth ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, December 1995.
- [5] Jeremy Fitzhardinge. Userfs—file systems implemented as user processes. <http://sunsite.unc.edu/pub/micro/pc-stuff/Linux-ALPHA/userfs/INDEX.html>, 1997.
- [6] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [7] M. Frans Kashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russel Hunt, David Mazières, Thomas Pickney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. Sixteenth ACM Symp. on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.
- [8] H. Richard Kendall, Vincent W. Freeh, Paul W. Schermerhorn, and Peter W. Rijks. Streaming extensibility in the modify-on-access file system. To appear in the *Journal of Systems and Software*.
- [9] H. Richard Kendall. The modify-on-access file system. Master's thesis, University of Notre Dame, Notre Dame, IN 46556, July 1998.
- [10] H. Richard Kendall and Vincent W. Freeh. The modify-on-access file system: An extensible Linux file system. In *Proc. of LinuxWorld Conference and Expo*, San Jose, CA, March 1999.
- [11] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. *ACM SIGOPS*, pages 1–14, December 1993.
- [12] A. David McNab. BSD portals for Linux 2.0. Technical Report NAS-99-008, NASA Ames Research Center, 1999.

- [13] Steve D. Pate. *UNIX Internals*. Addison-Wesley, 1996.
- [14] Gnu Hurd Project. Debian GNU/Hurd translators. <http://www.debian.org/ports/hurd/hurd-doc-translator>.
- [15] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach. An extensible I/O system. *Proceedings of 1986 Summer USENIX Conference*, pages 114–125, June 1986.
- [16] Dennis M. Ritchie. A stream input output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.
- [17] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [18] David S. H. Rosenthal. Evolving the vnode interface. *USENIX Summer Conference*, pages 107–117, Summer 1990.
- [19] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. Fourteenth ACM Symp. on Operating Systems Principles*, pages 203–216, Ashville, NC, December 1993.
- [20] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proc. 1999 USENIX Annual Technical Conf.*, June 1999.

Volume Managers in Linux

David Teigland
Heinz Mauelshagen
Sistina Software, Inc.
<http://www.sistina.com>

Abstract

A volume manager is a subsystem for on-line disk storage management which has become a de-facto standard across UNIX implementations and is a serious enabler for Linux in the enterprise computing area. It adds an additional layer between the physical peripherals and the I/O interface in the kernel to present a logical view of disks, unlike current partition schemes where disks are divided into fixed-size sections.

In addition to providing a logical level of management, a volume manager will often implement one or more levels of software RAID to improve performance or reliability. Advanced logical management tools and software RAID are the specialties of the Logical Volume Manager (LVM) and Multiple Devices (MD) drivers respectively. These are the two most widely used Linux volume managers today.

This paper describes the current technologies available in Linux and new work in the area of volume management.

1 Introduction

Managing large amounts of storage can be a difficult and sensitive task. Good management tools and infrastructure will allow more reliable storage systems to be deployed and administered. Volume managers have long been a cornerstone of storage subsystems. Specific features of current Linux volume managers will be discussed in detail with

a focus on how to use them. After a review of these current technologies, new developments in the field will be reviewed.

1.1 What is a Volume Manager?

Large file systems require the capacity of several disks, but most file systems must be created on a single device. A hardware RAID device is one solution to this problem. A hardware RAID device appears as a single device while in fact containing several disk drives internally. There are other excellent benefits of hardware RAID, but it is an expensive solution if one simply needs to make many small disks look like a single big disk. Volume managers are the software solution to this problem. A volume manager is typically a mid-level block device driver (often called a volume driver) which makes many disks appear as a single logical disk [Vahalia]. In addition to existing in the kernel's block I/O path, a volume manager requires user level programs to configure and manage partitions and volumes. The virtualized storage perspective produced by volume managers is so useful that often all storage, including hardware RAID, is controlled with a volume manager.

1.2 Linux Landscape

The Linux software RAID driver and the Linux Logical Volume Manager (LVM) are the two primary volume managers being used in Linux. This introduction will set the stage for further detailed descriptions of their features and how to use them in later sections.

At the end of the paper the reasons to choose one or the other should be clear.

In the simplest case a volume driver will logically append several disks one after the other as shown in Figure 1. For negligible additional cost, the driver can interleave the logical blocks among the lower level disks. Often called striping or software RAID-0, this technique provides no additional redundancy, but significantly increases the bandwidth of the logical device. In the same manner, the volume driver can implement higher RAID levels for increased redundancy. The Linux software RAID driver implements RAID levels 0, 1, and 5 [Vadala].

The management interfaces presented to the user is one emphasis of the Linux LVM [Mauelshagen]. LVM extends the concept of a single logical disk and provides *multiple* logical levels from which to manage storage. The additional abstraction levels allow logical *groupings* of storage to be manipulated independent of the actual logical devices which are being used. The user gains flexibility in allocating, moving, and replacing specific devices. Managing large and dynamic collections of storage hardware becomes much easier.

The latest feature of the Linux LVM is snapshotting. Snapshots provide an efficient way to backup an image of any file system built on an LVM logical volume.

2 Linux Kernel Interfaces

This section describes how volume managers are linked into the block device driver layer of the Linux 2.4 kernel. Those less interested in the kernel functions of volume managers may want to skip to the next section.

All read and write requests which enter the block I/O layer are broken into fundamental operations described by buffer heads. These buffer heads are the basic I/O descriptors used by device drivers. It is worth noting that buffer heads are no longer a significant caching mechanism. The page cache is now

the primary caching location. Buffer heads are still used, however, to describe I/O transfers to the page cache.

Once buffer heads have been created to describe a particular I/O operation, they are passed into the functions `submit_bh()` or `ll_rw_block()` to be queued for a mid-level driver. The specific mid-level driver (e.g. SCSI disk driver) is selected using the buffer head's "rdev" field which is the device number. The major number of rdev is used as an offset into a table of block device drivers. When a volume manager is used, the buffer head's rdev and rsector values specify a block location at the logical level. At the end of the process explained in the next section, the buffer's rdev and rsector values will be translated to describe a real physical block.

The mid-level driver periodically removes requests which have been placed on its queue and sends them to lower level host bus adapter drivers. The next section describes in more detail how requests are formed.

2.1 Block I/O Request Path

Each mid-level driver registered with the block device layer is defined in the `blk_dev[]` table by a `blk_dev_struct`. This structure provides a default `request_queue` for the driver as well as a method by which the driver can provide an alternate `request_queue`. The `request_queue` structure contains a number of important items. First, the head of the list of I/O requests for the driver is found here. Requests are taken from this list (and sent to low level drivers) after they've accumulated and some merging has been done with them. The functions used to merge and dispatch the queued requests are linked into the `request_queue` structure as well.

Between a buffer head (bh) entering the block I/O path and a request being dispatched from the mid-level driver's queue, the bh must be transformed into a "request structure" which is then added to the driver's queue. To do these operations, the driver's `request_queue` structure supplies a function called `make_request_fn`.

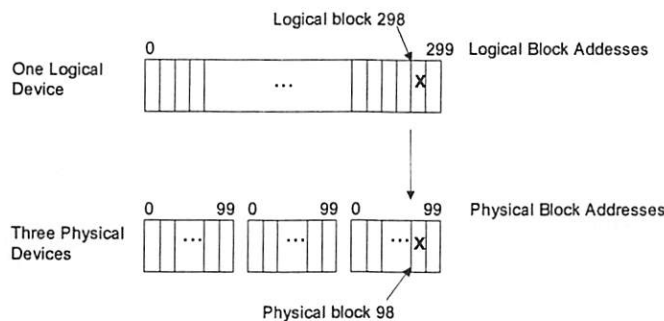


Figure 1: Linear Volume Mapping

As stated, `make_request_fn` will normally transform the `bh` into a request structure and then add the request to the driver's queue. In the case where the block driver is a volume manager, the `make_request_fn` simply rewrites the `rdev` and `rsector` fields of the `bh` to describe a real physical device and sector number. The function then returns a positive value indicating to the block I/O layer that the `bh` should be reprocessed. When the `bh` is reprocessed, a different driver with a new `make_request_fn` method is selected. This switch happens simply because the major number of the `bh`'s `rdev` changed.

The final `make_request_fn` routine will behave normally and create a request structure from the `bh` which is added to the request queue of the mid-level driver. Volume managers can be stacked on one another in which case the `bh` will be reprocessed multiple times.

The Linux 2.2 kernel provides no way for volume drivers to define their own methods. The old software RAID driver is called explicitly in the block I/O path when it is being used. The Linux LVM 2.2 kernel patches from Sistina Software add a generic mechanism to the block I/O path similar to the 2.4 kernel approach for volume managers to be inserted.

3 Linux LVM

Constructing flexible virtual views of storage with LVM is possible because of the well-

defined abstraction layers. User space tools used to configure each virtual level follow a similar set of operations. With these tools, online allocation or deallocation of storage to virtual groups is possible. Higher level virtual devices can then be expanded or shrunk online. Ultimately, a file system resizer is likely to be used.

The lowest level in the LVM storage hierarchy is the Physical Volume (PV). A PV is a single device or partition and is created with the command: `pvcreate /dev/sdc1`. This step initializes a partition for later use. On each PV, a Volume Group Descriptor Area (VGDA) is allocated to contain the specific configuration information. This VGDA is also kept in the `/etc/lvmtab.d` directory for performance reasons. The library functions of the LVM tools can access the local VGDA copy in a more time efficient manner than accessing all of the PVs.

Multiple Physical Volumes (initialized partitions) are merged into a Volume Group (VG). This is done with the command: `vgcreate test_vg /dev/sdc1 /dev/sde1`. Both `sdc1` and `sde1` must be PV's. The VG named `test_vg` now has the combined capacity of the two PV's and more PV's can be added at any time. This step also registers `test_vg` in the LVM kernel module and therefore it is made accessible to the kernel I/O layer.

A Volume Group can be viewed as a large pool of storage from which Logical Volumes (LV) can be allocated. LV's are actual block devices on which file systems or databases can be created. Creating an LV is done

by: `lvcreate -L1500 -ntest_lv test_vg`. This command creates a 1500 MB linear LV named `test_lv`. The device node is `/dev/test_vg/test_lv`. Every LV allocated from `test_vg` will be in the `/dev/test_vg/` directory. Other options can be used to specify an LV which is striped or has a physically contiguous allocation policy.

A Volume Group (VG) splits each of its underlying Physical Volumes (PV's) into smaller units of Physical Extents (PE's). One PE is typically a few megabytes but can range up to a few gigabytes to enable large disk subsystems. These PE's are then allocated to Logical Volumes (LV's) in units called Logical Extents (LE's). There is a one to one mapping between a LE and a lower level PE; they refer to the same chunk of space. A LE is simply a PE which has been allocated to a Logical Volume's address space.

A file system on `test_lv` may need to be extended. To do this the LV and possibly the underlying VG need to be extended. More space may be added to a VG by adding PV's with the command: `vgextend test_vg /dev/sdf1` where `sdf1` has been "pvcreated". The device `test_lv` can then be extended by 100 MB with the command: `lvextend -L+100 /dev/test_vg/test_lv` if there is 100 MB of space available in `test_vg`. The similar commands `vgreduce` and `lvreduce` can be used to shrink VG's and LV's.

When shrinking volumes or replacing physical devices, all LE's must sometimes be moved off a specific device. The command `pvmove` can be used in several ways to move any LE's off a device to available PE's elsewhere. There are also many more commands to rename, remove, split, merge, activate, deactivate and get extended information about current PV's, VG's and LV's.

3.1 Practical LVM Usage

There are many different situations in which LVM can be beneficial. Two different practical examples will be described in this section; one for small and the other for large systems.

3.1.1 Flexible Space Allocation

LVM can be very useful on small single disk systems and even laptops [Sistina]. Consider, for example, a system installation which is split among three file systems mounted at `/home`, `/usr`, and `root (/)`. If these three file systems are placed on three fixed partitions of the disk, a problem such as the following is common. The user or users of the system need much more space in their home directories than anticipated and the `/home` file system becomes full. No additional software has been installed on the system so the `/usr` file system has plenty of extra space. The fixed partitions prevent the `/home` file system from being expanded and the `/usr` file system from being shrunk.

The problem of file system space allocation could be solved easily if LVM was used. Using LVM, a single partition would be created on the disk and this partition would be made into a single PV. A volume group named *mainvol* would then be created from the PV. From *mainvol*, three LV's would be allocated, one for each file system. All the space need not be allocated from *mainvol* immediately, but can be saved and added later to the LV of the file system which needs space first. If all space from *mainvol* has been allocated to LV's, the allocation can be adjusted. To do this, one would shrink the file system which is underutilized, `lvreduce` the corresponding LV, `lvextend` the LV needing space, and finally expand the file system on the extended LV.

If a second disk was added to the system, the new disk could be added to the VG and used for growth in any of the file systems, or new LV's could be created for new file systems.

3.1.2 Volume Groups

In large systems with many storage devices the ability to dynamically adjust volume sizes as previously described is even more important. There are also other ways in which LVM can improve administration in large installations. Volume groups can be set up for dis-

tinct applications, business divisions, or other categories. This separation is useful because storage devices are often meant for a particular purpose.

Consider the following situation in which grouping storage devices into volume groups is important because of the different applications. A single machine is managing storage for both a mail server and a database server. The mail application requires two LV's for two file systems. The two LV's are allocated from a *mailvol* VG which is a pool of several individual SCSI disks. Disks are added and replaced in *mailvol* as demand changes. The machine also exports critical business databases on two LV's. These logical volumes are allocated from the *datavol* VG which is composed of two hardware RAID devices. It would be inappropriate for mail file systems to use space on the RAID devices, or for database logical volumes to use space on the individual SCSI disks. Separate VG's make it easy to ensure that an application will use storage space on the correct hardware.

3.1.3 Maintenance Tasks

LVM makes it simple to replace slow, small or aging disks in a VG while the LV's and file system's remain on-line. The `pvmove` command will move all logical extents off of a specified disk to free locations in the volume group on other devices. There are limitations to this when LV's are striped as well as problems if the system crashes during the `pvmove` operation which can potentially take a long time.

3.2 Logical Volume Snapshots

Snapshots are "frozen" images of a logical volume which can be used to back up the resident file system or database. A user command is used to initiate a snapshot at which point the current state of the LV is preserved and can be accessed through a special device node corresponding to the active LV's device node. A backup process uses the read-only "snapped" LV which does not change. The

file system or database can continue running while the backup takes place.

LVM uses a copy-on-write technique to allow continued updates to an LV while maintaining a previous frozen state. Any new writes result in a copy of the original block into a separate volume before the changed block can be written. The old and new locations of each modified chunk are maintained by LVM. Pointers to the original data are used when the snapshotted volume is accessed. The metadata managing copy-on-write mappings is persistent so snapshots remain available after reboots.

A snapshot LV requires only a fraction of the space of the original LV because only changed data needs new space. The actual size should be tuned according to the frequency of writes to the live volume and the life span of the snapshot volume. If write activity changes to the original volume after the snapshot has been taken, the snapshot size can be changed at run time. This is especially useful if too little space was allocated for the snapshot volume.

A snapshot is created as a new LV in the same volume group as the target LV. The volume group must have free space available to allocate the snapshot LV. The name of the snapshot LV can be assigned by the user just as the name of any new LV.

3.2.1 Application Consistency

Taking snapshots at a logical volume level is problematic if the file system, database, or user application is not in a consistent state at the moment the snapshot is taken. There are a variety of approaches to solving this problem depending on the particular application.

Databases often have hooks which can be used to quiesce them momentarily while a snapshot is taken. This will guarantee a consistent database state in the snapshot. A separate database specific program would likely coordinate the necessary database operations before and after the LVM snapshot is taken.

If a snapshot is simply taken of a file system volume, the resulting copy of the file system will look as it would if the system had crashed. When the snapped file system is then mounted read-only, it would appear to need an fsck in the case of a non-journalled file system or journal recovery in the case of a journalled file system. These operations (fsck or journal replay) require writing to the snapshot logical volume which is not allowed.

One solution to this problem is to unmount the file system before taking the snapshot and then remount the file system afterward. Requiring the file system or database to be taken off-line is not a widely accepted solution, however. Another approach would be to implement writable snapshots so recovery could be done when the snapshot file system is mounted. A third solution is to create a file system interface which a snapshot procedure could invoke to cause the file system to make its on-disk state consistent. In this case, recovery would not be needed.

Adding file system hooks for snapshots is the method being pursued and a kernel patch adds initial support for it. This approach still fails to address the issue of user mode applications which can be caught in an inconsistent state despite the file system itself being consistent. The snapshot could be taken between two writes from an application, both of which are required for the application to be consistent. This is a difficult problem because there are no standard API's to tell applications to make themselves consistent.

4 Software RAID

RAID (Redundant Array of Independent Disks) is a set of methods (or levels) for storing data on a set of disks to improve performance, reliability, or both [Patterson]. RAID levels 0, 1 and 5 are most common. RAID is often implemented in hardware controllers. A RAID controller appears to a host computer just as any other storage device would. Behind the hardware RAID controller is a group of disk drives. Depending on which RAID level the controller is configured for, it will

store data on the disks in different ways. The common RAID levels have the following characteristics:

- *RAID-0* improves performance by striping data blocks across the individual disks. Because all drives are ideally being used in parallel, the overall performance is the combination of the performance of each disk. Striping also reduces the chance that one disk will be subject to much more load than others (a "hot-spot"). There is no reliability or redundancy added by this level.
- *RAID-1* is also known as mirroring. The total usable storage space is half the physical storage capacity because half the disks are mirrored on the other half. Reliability is maximized because every disk has a duplicate. If a disk fails, its mirrored copy is synced to a spare or replacement disk. The device continues operating during recovery.
- *RAID-5* increases reliability allowing any one disk to fail without losing information. The space overhead is also low. Exclusive OR or "parity" values are calculated for each block stripe within the disk group. Blocks containing the parity values are interleaved among all the disks. If one disk fails, the data which was on that disk can be reconstructed using the data and parity blocks from the operational disks.

The RAID levels can also be implemented in the host's software on any collection of individual disks. RAID-0 (striping) and RAID-1 (mirroring) are the simplest to implement in a host driver and many volume drivers only support these two levels. Recovery procedures are the most difficult aspect of software RAID.

Performance of software RAID may be slower than hardware RAID for a couple reasons. Software RAID levels one and higher often require more data to be transferred between hosts and storage than would be required for hardware RAID. For example, the host must make two writes to separate disks

when maintaining mirrors whereas the data would be written only once to a hardware RAID device. In addition to the extra I/O to maintain parity, RAID-5 XOR calculations require extra CPU cycles.

One significant limitation in many software RAID drivers is the inability to extend RAID-0 (striped) or RAID-5 volumes. Data placement by either of these methods often is dependent directly on the total number of devices in the RAID set. Adding one additional disk would require shifting the location of all current data.

RAID-0 and RAID-1 are often combined to gain the advantages of both. They can be combined as RAID-0+1 which means that two volumes are mirrored while each volume is striped internally. The other combination is RAID-1+0 where each disk is mirrored and striping is done across all mirrors. RAID-0+1 may be considered less reliable because if each half loses any one disk, the entire volume fails.

4.1 Linux MD Driver Usage

The MD (Multiple Devices) software RAID volume driver has been entirely rewritten and vastly enhanced between the 2.2 and 2.4 kernels [Molnar]. In terms of logical devices, the MD driver provides the basic logical device abstraction without the management or snapshot capabilities of LVM. The focus of MD is high performance RAID and efficient background reconstruction in the event of disk failures. If extra devices are installed beforehand, they may be used as “hot-swappable” replacements by the driver.

There is much less administration in comparison to LVM because of the single device abstraction and the fact that most MD features are automatic. The primary step in creating an MD device is creating the logical definition in the `/etc/raidtab` file. Figure 2 shows a sample `raidtab` file. The contents of the file are:

- *raiddev* identifies which logical device is being described. The MD logical

```
raiddev /dev/md0
    raid-level      0
    nr-raid-disks   8
    persistent-superblock 1
    chunk-size      64
    device           /dev/sdb1
    raid-disk        0
    device           /dev/sdc1
    raid-disk        1
    device           /dev/sdd1
    raid-disk        2
    device           /dev/sde1
    raid-disk        3
    device           /dev/sdf1
    raid-disk        4
    device           /dev/sdg1
    raid-disk        5
    device           /dev/sdh1
    raid-disk        6
    device           /dev/sdi1
    raid-disk        7
```

Figure 2: A sample `/etc/raidtab` file.

devices are represented by `/dev/md0`, `/dev/md1`, etc.

- *raid-level* determines the level of software RAID MD will do for this device. The levels are 0 for striping, 1 for mirroring, 5 for parity RAID, or “linear” for no RAID.
- *nr-raid-disks* defines the number of disks within the logical device.
- *persistent-superblock* if set to “1” will make MD store configuration details (called a “superblock”) on each device. This allows auto-detection of MD devices by the kernel at boot time.
- *chunk-size* defines the stripe width when using RAID-0. It is the number of 1 KB units within a chunk, so 64 means data is interleaved at 64 KB boundaries.
- *device* specifies the device node of a partition assigned to the logical volume. There are as many *device* lines as *nr-raid-disks*.
- *raid-disk* follows every *device* line to indicate which sub-disk of the logical volume the *device* should be.
- *spare-disk* is used instead of *raid-disk* if the *device* is to be used as a replacement for any disk which fails.

The first time an MD device is set up, the command `mkraid` is used. The specific MD device (e.g. `/dev/md0`) follows the command on the command line. The `raidtab` file is read to find the device definition. A configured MD device can be enabled or disabled using the commands `raidstart /dev/md0` and `raidstop /dev/md0` [Østergaard].

5 GFS's Pool Driver

The Pool driver [Teigland] is a simple Linux volume manager which has been developed in conjunction with the Global File System [Preslan]. Three features of the Pool volume manager motivated by needs of GFS are mentioned here. In the future, it would be beneficial to integrate some of these features into another more widely used volume manager like LVM.

5.1 SCSI Mid-layer Interface

GFS is a shared disk cluster file system for Linux. Nodes in a GFS cluster must use a locking mechanism for synchronization. The first unique feature of Pool exists specifically to support GFS cluster locking. GFS can use a new SCSI command called the Device Memory Export Protocol, or DMEP, (still in the process of standardization) to access a pool of memory buffers implemented in the firmware of hardware RAID controllers and possibly disk drives [Barry]. When a Pool device is configured, the constituent real devices are tagged as supporting the DMEP command or not.

The Pool driver knows the physical distribution of DMEP buffers and presents a single logical pool of them to a GFS lock module (or other application). Primitive DMEP buffer operations like load and conditional store can be used to implement cluster wide locks ¹.

The Pool driver maps a logical DMEP buffer reference to a specific DMEP buffer on

¹GFS can use other distributed locking methods as well.

a real device and then sends the DMEP command to a SCSI target. To send the DMEP command, the Pool driver must bypass the standard SCSI driver APIs in the kernel designed for I/O requests and insert the DMEP Command Descriptor Block (CDB) in the SCSI command queue.

It would be possible to use GFS above a different cluster volume manager and continue to use Pool to access DMEP buffers. No regular data blocks would ever be read from or written to the Pool device and the size of the Pool could be zero. This would not require any changes to Pool and the cluster volume manager would not need any hooks for DMEP processing. This simple arrangement would be accomplished by creating a small partition on each DMEP capable device. These partitions would make up the Pool. The program which creates Pools writes a small label to the head of each Pool device, so the partitions must only be large enough to hold a Pool label.

5.2 Device Identity

The second feature of Pool is based on the knowledge that GFS and Pool will be used in a storage area network (SAN) or other environment where devices are shared directly among multiple hosts. A problem in SAN's are devices which change "identity" when the Fibre Channel loop or fabric is re-initialized. The volume manager can not assume that a physical device will always be matched with the same target address or device node.

To solve this problem, the Pool driver identifies Pool partitions by a label placed at the head of each Pool partition. At startup, Pool scans all the devices for specific Pool labels and when all the partitions have been found, the Pool can be activated. The labels impose a fixed ordering on the Pool partitions with respect to each other, but with no respect to physical ID.

5.3 Sub-Pools

The third feature of Pool is the ability to configure a Pool device with distinct “sub-pools” consisting of particular partitions or LUNs. In addition to common Pool device characteristics (like total size), the file system or mkfs program can obtain sub-pool parameters and use sub-pools in special ways. Currently, a GFS host can use a separate sub-pool for its private journal while the ordinary file system data is located in general “data” sub-pools. This can improve performance if a journal is placed on a separate device.

Performance can be further optimized because disk striping is configured per sub-pool. If many disks are used in a logical volume, striping among subsets of disks can be beneficial due to improved parallelism among sub-pools (e.g. four sets of four striped disks rather than 16 striped disks). Other possible uses for sub-pools or similar constructions in other volume managers are covered later.

6 Volume Managers on other Platforms

Volume managers have appeared in many different UNIX systems in many forms. This section will review a few of the major ones.

6.1 Veritas Volume Manager

The Veritas Volume Manager (VxVM) is a very advanced product which supports levels of configuration comparable to Linux LVM as well as the RAID levels of Linux software RAID [Veritas]. VxVM runs on HP-UX and Solaris platforms. Java graphical management tools are available in addition to the command line tools.

The abstraction layers of VxVM are more complex than those found in Linux LVM. The abstract objects include: Physical Disks, VM Disks, Subdisks, Plexes, Volumes, and Disk Groups [Veritas SAG].

The VxVM RAID-0 (striping) implementation allows a striped plex to be expanded. This is possible because of the way VxVM virtual objects are constructed. RAID-0 can be used with RAID-1 to combine the advantages of both.

The RAID-1 (mirroring) capabilities can be tuned for optimal performance. Both copies of the data in a mirror are used for reading and adjustable algorithms decide which half of the mirror to read from. Writes to the mirror happen in parallel so the performance slowdown is minimal. There are three methods which VxVM can use for mirror reconstruction depending on the type of failure. Dirty Region Logging is one method which can help the mirror resynchronization process.

The RAID-5 implementation uses logging which prevents data corruption in the case of both a system failure and a disk failure. The write-ahead log should be kept on a solid state disk (SSD) or in NVRAM for performance reasons. Optimizations are also made to improve RAID-5 write performance in general.

The VxVM “Hot-Relocation” feature is very similar to using hot-spares in Linux software RAID.

6.2 Sun Solstice DiskSuite

Sun’s Solstice DiskSuite (SDS) is a volume manager which runs on the Solaris operating system only [Sun]. SDS supports most of the VxVM features but is generally considered less robust than VxVM. The Metadisk is the only abstract object used by SDS limiting the configuration options significantly. The following list gives a basic comparison with VxVM.

- RAID-0 stripe sets cannot be expanded, although other stripe sets can be concatenated with existing sets to expand a volume containing striped devices.
- Proper RAID-1 mirror resynchronization is supported much like VxVM.

- Hot Spare disks can be configured.
- RAID-5 sets are supported, but can only be extended in the same limited fashion as stripe sets. RAID-5 reconstruction is a manual operation and does not include logging to guarantee data recovery. SDS does not include RAID-5 optimizations to improve write performance.

Both SDS and VxVM provide some level of support for coordinating access to shared devices on a storage network. This involves some control over which hosts can see, manage or use shared devices.

6.3 FreeBSD Vinum

Vinum is a volume manager implemented under FreeBSD and is Open Source like the Linux volume managers [Lehey]. Vinum implements a simplified set of the Veritas abstraction objects. In particular Vinum defines: a *drive* which is a device or partition (slice) as seen by the operating system; a *sub-disk* which is a contiguous range of blocks on a drive (not including Vinum metadata); a *plex* which is a group of subdisks; and a *volume* which is a group of one or more plexes. Volumes are objects on which file systems are created.

Within a plex, the subdisks can be concatenated, striped, or made into a RAID-5 set. Only concatenated plexes can be expanded. Mirrors are created by adding two plexes of the same size to a volume. Other advanced features found in the previous commercial products are not currently supported but are planned as a part of future development.

6.4 Linux LVM Relatives

IBM initially developed an LVM which was subsequently adopted by the OSF (now OpenGroup) for their OSF/1 operating system. The OSF version was then used as a base for the HP-UX and Digital UNIX operating system LVM implementations. The

abstraction model in this family of LVM's is different from the Veritas model. The Linux LVM implementation is similar to the HP-UX LVM implementation.

7 New Work in Linux

This section covers some more specialized topics and lists some areas of current and future research in Linux volume management.

7.1 Exporting Volume Metadata

The support for sub-pools in the Pool volume manager has already been mentioned. The general advantage of sub-pools is the ability to specify *sections* of a logical volume as having particular characteristics. The file system or database using the logical volume could take advantage of information (or metadata) obtained from the volume driver which matches qualitative information with block ranges. The information could be set permanently during configuration or it could represent statistical information gathered by the volume driver.

In the case of GFS, performance can be improved by simply placing different types of data in sections of the logical volume which map to different physical devices. This doesn't require any additional metadata than what a volume manager already uses. It simply requires a method of exporting this metadata to systems which can take advantage of it. If more detailed device characteristics were maintained by the volume driver, like I/O rates or reliability, the file system could be tuned even further.

One approach being considered for Linux LVM is adding an API for reading and writing exported chunks of metadata. The content of these chunks would be opaque to LVM itself. A file system or other application could use this exposed metadata for a variety of purposes including those already mentioned. The first difficulty is determining what volume geometry to export along with the meta-

data. Volume managers will have varying degrees of difficulty exposing useful volume geometry depending on the internal abstraction layers.

7.2 Cluster LVM

A Cluster LVM will coordinate operations in a shared storage environment. The future extensions of LVM will aim to implement a fully cluster transparent use of the existing command line interfaces and features. This will enable a user to run the familiar Linux LVM commands on an arbitrary cluster node.

In the first Cluster LVM release, a single global lock will allow only one LVM command to be run in the cluster at a specific point in time so that LVM metadata (VGDA's) are updated correctly. Cached metadata on all cluster nodes will also be made consistent. The global lock will be supported by a simple synchronous distributed locking API.

Later releases of the Cluster LVM will use a more fine grained locking model. This will allow pieces of the VGDA to be locked and updated individually. One of the advantages of fine grained locking is that the amount of metadata updated on all cluster nodes is minimized which reduces the latency of operations. Another advantage is that commands dealing with different LVM resources can run in parallel (e.g. creating two LV's in different VG's).

Some volume groups and logical volumes on the shared storage may be intended for a single node only. Other volumes may be for concurrent use as is the case when using a cluster file system like GFS. Metadata enhancements will be added to support different sharing modes on volumes.

7.3 Combining Volume Types

MD RAID volumes can be used along with LVM to gain the advantages of both volume managers. An MD device can be used as a Physical Volume within an LVM system. In

this case, whatever data is placed by LVM on the MD device will be more reliable as MD implements the RAID functions independently of LVM. The ordinary LVM tools can be used to manage the storage devices although the partitions within the MD volume could not be managed individually by LVM.

7.4 IBM EVMS

IBM's Enterprise Volume Management System (EVMS) is a volume manager project with the aim of producing a highly flexible system which could emulate the technical features and operational interfaces of nearly any other volume manager [Rafanello]. Each of the following abstraction layers in the EVMS architecture accepts "pluggable" modules which can deliver different features or interact with differing system types.

- *Device Managers* interact with the kernel's block device drivers to obtain basic information about what physical devices are connected to the machine. The bottom part of this layer must be very operating system and even kernel version specific. Device Managers present abstracted Logical Disks to the layer above.
- *Partition Managers* recognize specific partition types on Logical Disks. Rather than enforcing its own partition type on a disk, this layer allows Partition Managers to translate various disk partitions into abstract "Logical Partitions" which are used by the next layer.
- *Volume Group Emulators* merge Logical Partitions into Volume Groups. The Volume Group concept is slightly different on different platforms, so modules at this level create the behavior for the specific system of interest (Linux LVM, AIX LVM). Partition based volume managers which don't have the volume group concept can presumably be emulated as well.
- *Features* create usable Logical Volumes from Volume Groups. Different Feature modules will map and translate blocks differently between Logical Volumes and

Volume Groups. This is where software RAID can be performed or other operations like encryption. Feature modules can be layered above one another to combine their effects. Features are categorized or layered according to whether they operate at a volume level (e.g. encryption), at the aggregation level (e.g. software RAID), or at a partition level (e.g. bad block relocation).

- *File System Interface Modules* are not layers in the I/O stack like the others. Modules plug in here to interface with different file systems for the purpose of resize operations, or other things which the volume manager and file system need to communicate about.

8 Conclusion

This paper has described the current Linux volume managers and has reviewed the ongoing work to extend and improve them. Today, both the Linux LVM and Linux software RAID drivers are widely used. Those interested in maximal flexibility and manageability often use Linux LVM, especially when there are large numbers of devices to manage or large numbers of volumes to maintain. Those interested in high availability without the high cost of special hardware often use Linux software RAID in mirrored or RAID-5 configurations. Both products will continue to improve and simple methods to use them both together will likely emerge.

9 Acknowledgments

The authors would like to acknowledge the LVM group at Sistina Software for their input on this paper and the work on LVM. This includes: Patrick Caulfield, AJ Lewis, and Joe Thornber. We would also like to thank Matthew O'Keefe of Sistina for his input on this paper.

References

- [Barry] Andrew Barry, Kenneth Preslan, Matthew O'Keefe SCSI Device Memory Export Protocol, <http://www.sistina.com/gfs/Pages/dmep.html>
- [Lehey] Greg Lehey *The Vinum Volume Manager*, <http://www.vinumvm.org>
- [Mauelshagen] Heinz Mauelshagen, Primary author and maintainer of Linux Logical Volume Manager, <http://www.sistina.com/lvm>
- [Molnar] Ingo Molnar, Primary author and maintainer of Linux software RAID, <http://people.redhat.com/mingo/>
- [Østergaard] Jakob Østergaard, *The Software-RAID HOWTO*, <http://www.linuxdoc.org/HOWTO/Software-RAID-HOWTO.html>.
- [Patterson] D. Patterson, G. Gibson, R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the 1988 ACM SIGMOD Conference of Management of Data*, June 1988.
- [Preslan] Kenneth Preslan, Andrew Barry, Jonathan Brassow, Russell Cattlen, Andam Manthei, Erling Nygaard, Seth Van Oort, David Teigland, Mike Tilstra, Matthew O'Keefe, "Implementing Journaling in a Linux Shared Disk File System", *Proceedings of the Eighth NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Seventeenth IEEE Symposium on Mass Storage Systems*, March 2000.
- [Rafanello] Ben Rafanello, John Stiles, Cuong Tran, *Emulating Multiple Logical Volume Management Systems within a Single Volume Management Architecture* <http://oss.software.ibm.com/developerworks/opensource/evms/doc/EVMS.WhitePaper.1.htm>
- [Sistina] *LVM HOWTO*, http://www.sistina.com/lvm/doc/lvm_howto/index.html
- [Sun] *Solstice DiskSuite 4.0 Administration Guide*, <http://docs.sun.com>

[Teigland] David Teigland "The Pool Driver: A Volume Driver for SANs," Master's thesis, Dept. of Electrical and Computer Engineering, Univ. of Minnesota, Minneapolis, MN, Dec. 1999.

[Vadala] Derek Vadala "RAID on Linux", *Journal of Linux Technology*, Vol. 1, No. 2, April 2000, pp. 25-33.

[Vahalia] Uresh Vahalia, *UNIX Internals*, Prentice Hall, 1996.

[Veritas] *Features of VERITAS Volume Manager for Unix and VERITAS File System*, <http://www.veritas.com/us/products/volumemanager/whitepaper-02.html>

[Veritas SAG] *VERITAS Volume Manager System Administrator's Guide*, http://uw7doc.sco.com/ODM_VMadmin/sag-1.html

Linux LVM, Pool and GFS publications, HOWTOs and source code can be found at <http://www.sistina.com>.

The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX

Giuseppe Cattaneo Luigi Catuogno Aniello Del Sorbo Pino Persiano

Dipartimento di Informatica ed Appl.

Università di Salerno

Baronissi, SA, Italy, 84081

cattaneo@unisa.it

luicat@dia.unisa.it

anidel@dia.unisa.it

pino.persiano@unisa.it

Abstract

In this paper we describe the design and the implementation of the Transparent Cryptographic File System (TCFS). TCFS is a cryptographic distributed filesystem (à la NFS). It lets users access sensitive files stored on a remote server in a secure way. It combats eavesdropping and data tampering both at the server and over the network by using encryption and message digest. Applications access data on a TCFS filesystem using the regular systems calls thus yielding complete transparency to the users.

TCFS implementations for Linux, NetBSD and OpenBSD are available at <http://www.tcfs.it>

1 Introduction

Today's networking makes it feasible to share resources over a network. Filesystems have been historically one of the first services to be distributed over a network (see Sun's NFS [13]). Nowadays these services are even more necessary. In fact the wide spread of mobile equipments (such as PDA or laptop computers) strongly require the availability of a com-

mon file repository accessible from any place all over the world with different network strategies. Distributing applications and services over a network offers obvious advantages but creates several security problems: unauthorized users might gain access to restricted services.

Within the context of distributed filesystems this phenomenon is easily seen. In a distributed filesystem, we have two types of actors: servers which have direct access to a local filesystem and clients that wish to access files on filesystems local to the server. Servers and clients are connected through a communication network. Let us take NFS as an example. NFS is very naive in its approach to security. Roughly speaking, the server receives requests for blocks of data from the client and sends the data block in clear over the network. It is a simple task to eavesdrop over the conversation and thus read the data [9]. Moreover, the access to the data is granted by the server on the basis of the uid (and gid) communicated by the client. Thus nothing prevents a pirate client from giving the server "the right" information and thus gaining access to the whole file system.

Users who wish to protect their files should

adopt measures to prevent exposure of sensitive data. This problem can be addressed at several levels: user, application and system level. In this paper we present the Transparent Cryptographic FileSystem (TCFS, in short) that addresses the problem of securing data in a distributed filesystem at the system level. The TCFS project started in 1995 but only in the last 12 months the experimental analysis have demonstrated its efficiency and its robustness.

Before describing the features of TCFS, let us present arguments in favor of a solution at the system level as opposed to solutions at the user or application level.

Several tools exist to encrypt the content of files and directories. However, we point out that this approach suffers of two main drawbacks:

1. Ease of use. Data reside in encrypted form on the filesystem. Before accessing the data, the user needs to decrypt it before and, after he has finished, he needs to re-encrypt the data. This is very cumbersome and users would tend to avoid this step. In general, a well-know security practice principle states that security has to come to little or no operative cost to the user.
2. Network. Encrypting and decrypting data in a distributed filesystem does not guarantee that the data is not exposed to an unauthorized party. Indeed, once the user has decrypted the data, it is stored unencrypted on the server. Thus data is leaked to the filesystem server. Moreover, data is transferred between the server and client in clear and thus can be read by eavesdroppers.

Several widely used applications offer an encrypting service: when data is saved to disk, the user can choose whether to encrypt it or not. This approach addresses the usability

problem but data is still vulnerable when it travels on the network.

2 Related Work

Distributed file systems have been the focus of much research in the last decades starting with the early proposals [3, 13].

Along this line of research, transparency (the fact that the filesystem is not local should be hidden from applications and users) and security have been two main issues. Some systems, like Compaq's Cluster File System[4], do guarantee transparency even in presence of faults, but assume that confidentiality of data and authentication of the parties involved is achieved through some other system. The Andrew File System [6] and CODA [14] instead provide mechanism to authenticate servers over public lines and to ensure that client-server communication could not be eavesdropped. The serious key-management issues arising in such file systems is addressed by the Self-certifying FileSystem [8] (more on SFS in Section 2.3). We do point out that both AFS and CODA assume the server to be trusted and store the data in clear on the server machine. TCFS instead stores the files in encrypted form thus denying the server access to the data in clear.

In this section we review some of the work present in the literature stressing the differences with TCFS either in implementation or in architectural design.

2.1 Cryptographic File System

Matt Blaze's Cryptographic File System (CFS)[2] is probably the most widely used secure filesystem and it is the closest to TCFS in terms of architecture. CFS encrypts the data before it passes across untrusted components, and decrypts it upon entering trusted components. CFS users create directories associated

with keys and each file created in a protected directory is automatically encrypted.

CFS simulates a remote NFS server which exports on demand encrypted directories. All operations performed in clear by the user on a protected resource are mapped by CFS to the source directory (created by `cmkdir`) encrypted. During (and after) the user session, an intruder could not obtain clear data from the source directory.

CFS, that was the primary motivation of the work presented in this paper, presents the following characteristics.

- CFS is not transparent to the user. Encrypted directories have to be explicitly attached to a specific directory by the user before they can be accessed.
- Cryptography granularity is at the level of the directory. This implies that the user must remember a password for each encrypted directory she owns. Moreover, *all* files in an encrypted directory are encrypted as opposed to TCFS where the user can choose which files to keep in encrypted form and which to keep in clear.
- CFS has been implemented as a user application. On the positive side, this approach makes it very easy to port CFS to different operating systems. On the negative side, this increases its vulnerability to attacks to the client machine and reduces its performance.
- CFS does not allow group sharing of protected resources nor it offers data authentication.

2.2 CryptFS

CryptFS[18] is a cryptographic file system implemented at the virtual inode level using the abstraction of Stackable File Systems [5] and can be used on top of local or remote file systems. Like TCFS it uses the cipher block

chaining encryption mode within a block (usually 4k or 8k long) and only provides Blowfish as encryption algorithm.

CryptFS is part of the FiST (File System Translator) [19] project developed by the same authors. FiST is a system that uses a high-level language to describe a file system and to generate the working implementation for the target operating system, thus improving portability.

We found no source code for CryptFS, so we could not compare it with TCFS. A performance comparison between CryptFS and (an older version of) TCFS is found in [18]. CryptFS does not ensure data integrity and does not allow unencrypted files on an encrypted file system. This has a non trivial impact on the performance as, for example, CryptFS needs not to check if the file is clean or encrypted, nor it needs to check the integrity of blocks upon reading. We also stress that there is no support for threshold group sharing of encrypted files.

2.3 Self-certifying File System

The Self-certifying File System (SFS)[8] addresses the issue of key management in cryptographic filesystems and proposes separating key management from file system security. Servers have a public key and clients use the server public key to authenticate the server and establish a secure communication channel. To allow clients to authenticate servers on the spot without even having heard of them before, SFS introduces the concept of a “*self-certifying pathname*.” A self-certifying pathname contains the hash of the public-key of the server, so that the client can verify that he is actually talking to the legitimate server. Once the client has verified the server a secure channel is established and the actual file access takes place.

Remote SFS file systems are accessed through the `/sfs` mount point. An SFS pathname obeys the following syntax:

/sfs/location:hostid/real/pathname, where "location" is the name (IP address or DNS Name) of the server exporting the file system and "hostid" is the hash of a string containing the server's public key and some other information. SFS does not care on how the pathname has been obtained by the user; a user can eventually obtain hostid's using an existing PKI (Public Key Infrastructure). On the other hand, once a self-certifying pathname for the files he is interested in has been obtained, users do not need to remember any key.

3 The TCFS Architecture

TCFS relies on a very simple architecture. Data is stored in encrypted form on the server filesystem. Each time an application running on a client has to read data, the client kernel requests the appropriate block of data from the server. The server ships the block of data in encrypted form to the client. The client decrypts the block of data before passing it to the application. A write operation is accomplished in a similar way. Suppose a client application wishes to write data on a filesystem. The application passes the data to the client that encrypts the data and passes it to the server. The server, upon receiving data from the client over the network, stores the data on the filesystem.

This architecture has several advantages:

- Minimal trust model. The TCFS architecture does not rely on the the server nor the network being trusted. In fact, the server only sees encrypted data and data travels over the network only in encrypted form. As we will see when we discuss the implementation details, the client can detect any unauthorized modification of data. Of course, since clients can access data only through the servers, TCFS cannot prevent servers from erasing the data or from denying access to the clients. All

the encryptions and decryptions are performed by the client on which the application is running. Thus the application and the user have to trust the client kernel used to access the filesystem. This is not a serious limitation for cases in which users employ personal workstations to access files.

- Low system administration impact. TCFS does not require any additional duty to the system administrator of the server. All filesystem maintenance operations on the servers need not to know about TCFS. Actually, the system administrator himself might ignore that his local filesystem is actually a TCFS filesystem.
- Low impact on client applications. TCFS was designed to reduce the impact on the applications. Client applications access files on a TCFS filesystem through the usual system calls and thus they need not to be re-written or re-compiled to work with TCFS. Client applications need not to deal with key management.
- Low impact on the user. Besides issues regarding key management, TCFS has little or no impact on the final user. She can still access her files using the same applications and ignore completely that the files she is accessing are stored on a remote server in encrypted form. TCFS guarantees to the users and to the applications a level of transparency similar to NFS. Nonetheless, TCFS provides users needing a greater control on the encryption/decryption policy, the ability to control which files are encrypted and which are not.

3.1 Authenticating servers

TCFS assumes a very minimal trust model: the user only needs to trust the client machine

used to access the TCFS filesystem. We point out that this is a very minimal assumption as it is very hard to conceive a system that preserves security even in presence of untrusted client machines.

On the contrary, a user needs not to trust the server on which the filesystem physically resides. Indeed, the server only has access to data in encrypted form which is of no use. Obviously, the server can modify the data stored and there is nothing that the user can do to prevent that. However, since TCFS includes authentication mechanisms for the data, if the server modifies the data, the user will immediately notice that data has been altered.

Similarly, there is no need for the client to authenticate the server. Suppose that a pirate host has managed to impersonate the legitimate TCFS server. We stress that, even in this case, the privacy of the user is not compromised. Indeed if the client tries to write, then the private server only gets encrypted data. On the other hand, if the client performs a read operation, the data he/she will receive from the server will not be authenticated and thus immediately rejected by the client.

4 Key management

In the design of TCFS we have decided to keep key management issues separated from the actual cryptographic filesystem. In the two implementations of TCFS for Linux and BSD-like kernels, TCFS provides a simple interface to pass key to the kernel (by ad-hoc `ioctl` calls, or by upgrading the filesystem mounting). On top of this basic key-management primitive more sophisticated key management schemes can be built. As part of the TCFS project we have implemented three key management schemes that we termed the Raw, the Basic and the Kerberized Key Management Scheme that we briefly review in the rest of the section. TCFS can perform key management at different levels: at the process level in

the sense that each process has its own key to access the TCFS filesystem; at the user level in the sense that each user has its own key and all processes with the same uid use the same key. Moreover, TCFS provides a simple threshold mechanism for sharing files in a group of users.

4.1 Group Sharing

TCFS includes the possibility of threshold sharing files among users. Threshold sharing consists in specifying a minimum number of members (the threshold) that need to be “active” for the files owned by the group to become available. TCFS enforces the threshold sharing by generating an encryption key for each group and giving each member of the group a share using a Threshold Secret Sharing Scheme [15]. The group encryption key can be reconstructed by any set of at least threshold keys.

A member of the group that intends to become active does so by pushing her/his share of the group key into the kernel. The TCFS module checks if the number of shares available is above the threshold and, if it is so, it attempts to reconstruct the group encryption key. By the properties of the Threshold Secret Sharing Scheme, it is guaranteed that, if enough shares are available, the group encryption key is correctly reconstructed. Once the group encryption key has been reconstructed, the files owned by the group become accessible. Each time a member decides to become inactive, her share of the group encryption key is removed. The TCFS module checks if the number of shares available has gone under the threshold. In this case, the group encryption key is removed from the TCFS module and files owned by the group become inaccessible.

The current TCFS implementation of the group sharing facility requires each member to trust the kernel of the machine that reconstructs the key to actually remove the key once the number of active users goes below the threshold. Future implementations will re-

move this requirement by performing the reconstruction of the key in a distributed manner.

4.2 Raw Key Management Scheme

TCFS provides a simple interface for users applications to pass keys to the kernel which we call the Raw Key Management Scheme (RKM, in short). By using the RKM API, an application can provide the key to the TCFS kernel. Subsequently, the TCFS kernel will use the key provided to perform encryptions and decryptions. No check is performed by the TCFS kernel on the key and the application has to make sure that the right key is passed to the kernel. The RKM scheme is not intended for the end user but only as a basis on top of which to build more sophisticated KM schemes.

4.3 The Basic Key Management Scheme

This scheme allows users to generate their keys and to store them in a database in encrypted form using the login password as key. Thus, TCFS users must not remember their master key, but only their login password. To benefit of the BKMS a user must be registered with the key database (typically the file `/etc/tcfspwdb`) by the system administrator. The usage of the BKM scheme follows the phases below:

1. The system administrator registers a user to the key database (Fig. 1) by issuing the command `tcfsadduser`.
2. The user creates his master key by running the `tcfsngenkey` command. `tcfsngenkey` generates a random key, encrypts it with the user's password, and stores it in the entry of the key database associated with the user.

3. When the user needs to access his encrypted files, he must extract his master key from the database (providing his password), and give it to the TCFS layer. This operation can be performed with the `tcfsputkey` command (Fig. 2).
4. The user terminates his session by running the `tcfsrmkey` command which erases the key from the kernel.

Setting up a TCFS group requires the following steps:

1. The system administrator creates a normal UNIX group, then creates a TCFS group by running the `tcfsaddgroup` command. This utility asks for the number of group member, the threshold, the password, and the username of each member of the new TCFS group. For each member, a share is created, encrypted with the password of the respective user and then it is stored in the TCFS group keys database (`tcfsgpddb`).
2. To become active, a member of a TCFS group pushes her share into the kernel. This can be accomplished by executing the command `tcfsputkey` with the `-g` switch. Note that, user can get access to shared files only if the number of the same group shares pushed to the kernel is greater or equal to the group's threshold.
3. The `tcfsrmkey -g` command ends the user's session.

The aim of the BKMS is to provide the user with a simple to use management scheme. It is not to be considered very secure as the user master key is protected by the user login password that can be compromised in several ways.

4.4 The Kerberized Key Management Scheme

Kerberos is a distributed authentication service developed in the late 80s at M.I.T. [17].

root# tcfssadduser	
Username to add to TCFS database: jack	
Ok	<i>now jack has an empty entry in the key db</i>
	<i>before to have his first TCFS session, jack must run:</i>
jack\$ tcfsgenkey	
Insert your password, please:	<i>give his login password</i>
Press 10 random keys, please: *****	<i>seed</i>
Key succesfully generated.	<i>now jack's enty in the key db contains his master key, ecrpyted with his login password</i>
	<i>whenever superuser must remove jack from TCFS key database, he must run:</i>
root# tcfssrmuser -u jack	

Figure 1: Creating a TCFS user with the BKM Scheme

Kerberos requests as a trusted third-party authority that provides authentication to all the actors in a distributed environment. Kerberos makes possible for a client and a server to authenticate each other and to establish a private communication channel. The Kerberized Key Management (KKM) Scheme provides a strong alternative to the BKM Scheme. We introduce a new component: the *TCFS key server* (TCFSKS) that maintains a database of master keys. Clients (*i.e.*, kerberized TCFS utilities such as `tcfspukey`, or *TCFS-aware* applications) authenticate themselves on Kerberos, and obtaining a session key and a ticket, send to TCFSKS the requests (for example: get the user key or store a new key) over the network. Administrative operations, such as adding/removing users and group, can be performed in the same way. Since no changes have been made to the interface of front-end utilities, an user does not feel any difference between Kerberized and Basic Key Management procedures. The only substantial difference is that now all Key Management operations are performed over the network and thus several TCFS clients can share the same TCFS key database (in the BKMS the key db is local to

the client).

Communication among client and TCFSKS follows these steps:

1. At the end of the Kerberos authentication, the client obtains the session key.
2. The client sends its request and its ticket to the TCFSKS.
3. The server decrypts the message and sends back the response and the ticket.
4. The client get the response and discards the ticket.

5 Cryptographic Engine

TCFS does not employ a fixed encryption scheme but for each file a different encryption engine can be specified. Encryption engines need to conform to a specific interface and, in the Linux implementation, can be kernel loadable module. Modules for all the major encryption scheme are provided with the implementations. Having modular encryption, allows user to plug into TCFS its own encryption module for increased user security. Modu-

jack\$ tcfsputkey -m /mnt/tcfs	Jack starts his session
Password:	giving his login password
	now, Jack can encrypt/decrypt and access
	transparently to encrypted files.
jack\$ cd /mnt/tcfs	
jack\$ echo "Hello World!" > first	the file "first" is still in clear
jack\$ tcfsflag +X first	toggles first's cryptographic flag
	now it is stored encrypted
jack\$ cat first	all standard application can access
Hello World!	encrypted files
	while Jack's key is available to the kernel
jack\$ cp first second	can be read,
	copied and so on..
	the file "second" is stored in clear
jack\$ tcfsrmkey -p /mnt/tcfs	Jack removes his master key from the kernel
jack\$ cat first	
permission denied	since the master key has been removed,
	access to encrypted files is not
	allowed.
jack\$ cat second	
Hello World!	second is still in clear, TCFS session
	has no effect on clear files

Figure 2: A simple TCFS session

lar encryption allows users and system administrator to pick their favorite block encryption scheme. Thus, for this section we denote by $E(\cdot, \cdot)$ and $D(\cdot, \cdot)$ the encryption and decryption algorithm associated with the encryption scheme actually employed (see Figure 3). The size of the block encrypted by the encryption need not to be equal to the block size of the file. Each file has a header that contains some information about the file itself (e.g., TCFS version number, cipher id).

Each user A is associated with a *master key* K_A as described in Section 4. For each file f a *file key* K_f is randomly chosen. The file key is encrypted using the master key of the user and stored in the file-key field of the header. Each block of a TCFS file consists

of two parts: the data and the authentication tag. Each block of an encrypted TCFS file is encrypted with the encryption algorithm E in CBC mode using a different *block key*. Block of unencrypted TCFS file are stored in clear. The block key is computed by applying the hash function to the concatenation of the file key and the block number. The authentication tag of an authenticated TCFS file is computed by hashing the concatenation of the block data and the block key. On the other hand, unauthenticated TCFS files have an authentication tag consisting of NULL bytes.

This way of encrypting and hashing the blocks exhibits the following security characteristics:

1. If a robust encryption scheme is used then

encrypted files cannot be read without knowledge of the file-key or of the user master key.

2. Since each file is encrypted using a different file-key, it is not possible to check whether two encrypted files correspond to the same cleartext.
3. Moreover, since each block is encrypted using a different block key, it is not possible to check whether two blocks of the same file correspond to the same cleartext.
4. The authentication tag is to prevent that data on the server is modified. Obviously, since TCFS servers have physical access to the filesystem there is nothing that could prevent the server from modifying files. Our goal is thus to guarantee the user that no modification will go unnoticed. Our authentication mechanism guarantees:
 - (a) Modification of a block without recomputing its authentication tag is easily detected by the TCFS client. However, recomputing the authentication tag of a block requires knowledge of the block key which in turn depends on the file key which is encrypted using the user master key.
 - (b) Since each block authentication tag also depends by the block offset without knowledge of the block key it is not possible to insert foreign blocks of data or to swap two blocks of the same files.Moreover, since the authentication tag depends also on the file key, it is not possible to import block from other files.

6 Implementations

TCFS is designed to work in kernel space as an intermediate layer between the Virtual File System (VFS) and the storage file systems (such as EXT2FS, UFS, NFS and so on). In this way, user applications can perform all the usual file operations by means of system calls interface, without being rewritten/recompiled. Users can perform protection/encryption operations with apposite utilities which interact to the TCFS layer by the `mount` and `ioctl` system calls.

The TCFS layer only touches application data, and not file system logical structures (such as inodes, directory organization, etc.). Hence, although protected files result incomprehensible to the server's system administrator, all the disk maintenance tasks (check, backup, recovery), can be performed as usual.

TCFS has been implemented on Linux, and {Net,Open}BSD. The two versions are quite different, due to the different characteristics of the respective operating system.

All key management features (excepting pushing/removing keys) have been implemented at user level. To make easy the development of *tcfs-aware* application, or further key management schemes implementation, TCFS is provided of a development library that includes several functions which allow to manage every aspects of interaction among user and TCFS hiding any OS-specific implementation details.

6.1 The Linux version

The Linux version of TCFS consists in an extended NFS client with cryptographic features; a remote NFS server acts as the *storage file system*.

The client host mounts the remote filesystem as a TCFS filesystem (by means of a patched version of `mount`). Client and server communicate by means of NFS protocol. The user provides the encryption key to the TCFS

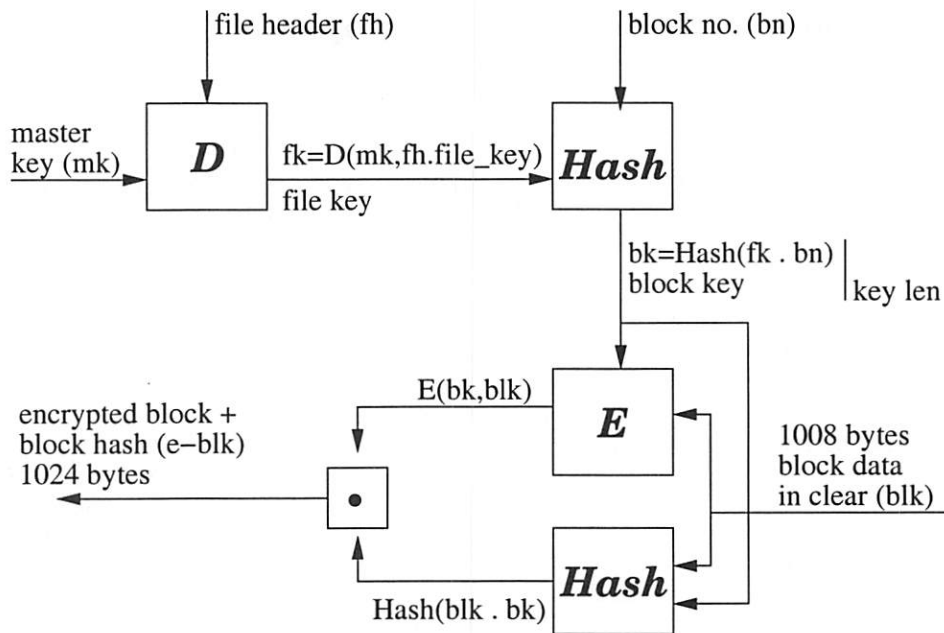


Figure 3: Encryption of blocks in TCFS

client so to allow encryption and decryption of files. The TCFS server instead is a simple NFS server: it needs not to know that the exported filesystem is indeed a TCFS filesystem. This has the advantage of making possible to use TCFS on any host that runs an NFS server.

User-level applications passes keys and directives to TCFS by calling the system call `ioctl` on the filesystem's mount point; for this purpose, TCFS introduces some new `ioctl` commands.

Since version 2.0, the Linux kernel features Loadable Kernel Modules (LKM) (see [1] and [11]). LKMs consist in some parts of the kernel (usually device drivers or filesystem switches) which can be loaded at runtime whenever they are needed. Cryptographic services, in TCFS, are implemented as kernel modules. Thus, ciphers can be selected at runtime by loading the proper dynamic kernel module. Users can choose different ciphers to encrypt/decrypt different files or directories. Each cipher mod-

ule is completely independent from the others and it is possible to load, unload different engines or even to use simultaneously several cryptographic engines.

The Linux TCFS implementation allows to have client and server running on the same host. However, the communication between the two takes place using the NFS protocol. This has the drawback of slowing down the communication.

6.2 The BSD version

An operating system based on 4.4BSD[10] provides to the kernel a generic interface to several kinds of file systems: the virtual-node (vnode) layer. The vnode layer features an object-oriented interface which abstracts the invocation to filesystem-specific operations, implemented at an underlying level. This makes possible to mount and access different kind of file system in the same way. 4.4BSD file

system switch has been provided of a mechanism for stacking filesystems on top of one other (proposed by Rosenthal[12], and refined by Heidemann and Popek[5]). The bottom of a filesystem stack is usually a storage filesystem (which directly interacts with the device driver). The layers above, can implement on their own any functions and/or redirect them downstairs (with or without argument transformations).

TCFS for BSD has been implemented as a file system layer. This approach presents an important advantage: since the cryptographic layer can be mounted upon any filesystem by mean of vnode interface, encryption of local filesystems (improving TCFS performances on these ones) does not need the NFS to introduce the cryptographic file-operators. Furthermore, TCFS for BSD has been developed by writing only those operators which required the introduction of cryptographic services whereas other calls have been redirected to undelying filesystems.

Users send their directives to TCFS by updating the mount-point parameters. The TCFS layer adds to the usual arguments of the mount system call two new sets of data that cointain directives and their arguments:

1. The data concerning the user's directive: the command, the key, the key's owner, etc.
2. A set of data which represents the status of the filesystem: number of active keys, error codes, information about the cipher.

TCFS flags and optional attributes management is performed by means of some `ioctl` calls.

6.3 Process keys

The BSD implementation of TCFS makes possible to provide different keys to different processes belonging to the same user. Thus users can work with several keys simultaneously

and, moreover, he can setup batch jobs which works on encrypted resources. User applications do not need to be rewritten/recompiled and, the process keys management is completely transparent. To make easy to run application with different keys, we developed the `tcfsrun` utility. This utility asks the user for the process key, passes it to the kernel and withdraws it when the user application ends. All kinds of keys (user, group and process) are managed independently, so, user can use his masterkey and group-shares normally even while any applications work with their own key.

7 Performance

In this section we present the overall TCFS performances analysis as a proof of concept.

We have employed a modified version of the Andrew benchmark [7]. The benchmark takes as input a subtree containing the source code of a UNIX application (in our case we used the sources of the GNU `make` application). The benchmark consists of five phases:

1. *Directories creation*: the source directory hierarchy is reproduced several times into a target directory on the tested file system.
2. *File copy*: all files of the source directory are recursively copied to a directory of the target subtree.
3. *Recursive directories stats*: attributes of each file in the target subtree are recursively scanned.
4. *Recursive files scan* : recursive reading of files in the target subtree.
5. *Compilation*: files on target subtree are compiled and linked.

We have performed four suites of tests. The first suite measured the performance of NFS.

In the second suite of tests, we measured the performance of TCFS on files that are not encrypted. Thus, input/output is still performed by TCFS but no encryption engine is invoked. The last two suites deal with TCFS in which encryption is performed by the NULL module (encryption using the identity function) and the 3DES module (encryption using Triple-Des[16]). TCFS with the NULL module differs from the second test suite as here an encryption engine, albeit a trivial one, is invoked.

All tests have been performed on a Pentium II at 233MHz with 64Mb of memory. We have performed two series of experiments. In the first series the results obtained were more influenced by the performance of write operations as we had the source tree on a local filesystem and the destination files on the remote filesystem (TCFS with 3DES, TCFS with Null, TCFS without encryption and NFS). In the second series of experiments we did the opposite: the source tree was on the remote filesystem and the destination filesystem was a local filesystem. Thus, the measurements were mainly affected by the performance of a read operations.

The figures reported are the average of 10 runs with the client and the server running on the same machine so that network latency is not an issue in the measurement.

As it is obvious from the experimental data, reading is much faster than writing. This is due to the fact that, unless a whole new block is written, a write operation involves reading a block, decrypting it, modifying it and re-encrypting it. Moreover, TCFS does not perform very well at random accessing encrypted files. Since TCFS encrypts each block using CBC, reading one byte might involve decrypting a whole block considerably slowing down the operation. This however does not have to be considered an inherent limitation of TCFS as it only depends on the specific cryptographic engine employed. Much faster random access can be obtained by encrypting in ECB (Electronic CodeBook) mode even

though in this case the confidentiality of the data is considerably weakened.

The overhead introduced by TCFS can be seen by comparing the first column with the second and the third column of Figure 4. As it can be seen, TCFS NONE exhibits performances very similar to NFS. More surprisingly, TCFS NULL is much slower than NFS. This is due to the following phenomenon. TCFS forces the remote attribute checking before each read/write operation whereas NFS does not. We expect that removing this check would have no impact on the security and keep the performance of TCFS NULL test much closer to NFS.

TCFS 3DES is the slowest of all and this is mainly due to the time to perform encryption/decryption. Indeed the difference between TCFS 3DES and TCFS NULL is exactly the overhead introduced by the cryptographic engine. Reducing this gap calls for a better cache management strategy, an issue that at the moment has not been considered yet. Also, we stress that when this test were performed with client and "remote" filesystem residing on the same machine. On a loaded Ethernet, the encryption/decryption overhead is likely to be absorbed by the network latency.

We believe that TCFS still has room for improvement (we would like to see TCFS NULL closer to TCFS NONE) but at the moment its performances are acceptable.

8 Acknowledgments

The authors thank Angelo Celentano, Andrea Cozzolino, Ermelindo Mauriello and Raffaele Pisapia that were part of the first TCFS team and took part in the initial stage of the TCFS project.

Exporting DATA to remote filesystem				
Phase	NFS	TCFS NONE	TCFS NULL	TCFS 3DES
Creating directories	0.109	0.178	0.648	0.698
Copying files	1.385	2.777	9.047	15.924
Recursive directory stats	2.215	4.798	5.558	6.537
Scanning each file	3.074	7.489	10.047	16.129
Compilation	36.802	57.791	1m3.874	1m27.929
Importing DATA from remote filesystem				
Phase	NFS	TCFS NONE	TCFS NULL	TCFS 3DES
Creating directories	0.052	0.065	0.081	0.093
Copying files	0.282	1.545	2.548	5.462
Recursive directory stats	1.355	1.449	2.273	2.388
Scanning each file	2.261	2.464	4.038	4.267
Compilation	34.634	36.653	35.448	48.364

Figure 4: Performance of the TCFS Linux implementation as measured with the Modified Andrew Benchmark

References

- [1] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, *"Linux Kernel internals"* Addison-Wesley, 1996
- [2] M. Blaze, *"A Cryptographic File System for UNIX"*, First ACM Conference on Communication and Computing Security, pp. 158-165, Fairfax VA 1993.
- [3] C. T. Cole, P. B. Flinn, A. B. Atlas, *An Implementation of an Extended File System for UNIX*, USENIX Conference Proceedings, Summer 1985, pp 131-149, Portland OR.
- [4] *"Cluster File System in Compaq TruCluster Server"*, Compaq White Paper, Unix Software Division, Compaq Computer Corp, August 2000.
- [5] J. S. Heidemann, G. J. Popek, *"File System development with stackable layers"*, ACM Transactions on computer systems, vol 12, no. 1, pp. 58-89, February 1994.
- [6] J. H. Howard, *"An overview of the Andrew File System"*, Proceedings of the USENIX Winter Technical Conference, Feb. 1988, Dallas TX.
- [7] J. H. Howard, M. L. Kazar, S. G. Meneses, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West, *Scale and performances in a distributed file system*, ACM Transaction on Computer System, Feb. 1988.
- [8] D. Mazières, M. Kaminsky, M. F. Kaashoek, E. Witchel *"Separating key management from file system security"*, Proceedings of 17th ACM Symposium on Operating System Principles (SOSP '99), Kiawah Island, South Carolina, December 1999.
- [9] S. McCanne, V. Jacobson, *"The BSD Packet Filter: a new architecture for user-level packet capture"*, Proceedings of the 1993 winter USENIX conference, pp. 259-269, San Diego CA, 1993.
- [10] M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarterman, *"The Design and Implementation of the 4.4BSD Operating System"*, Addison Wesley, 1996

- [11] O. Pomerantz, "*Linux Kernel Module Programming Guide*", GPL licensed book, 1999
- [12] D. Rosenthal, "*Evolving the vnode interface*", USENIX Association Conference Proceedings, pp. 107-118, June 1990.
- [13] R. Sandberg, D. Goldberg, S. Kleimann, D. Walsh, B. Lyon, "*Design and implementation of the Sun Network Filesystem*", USENIX Association Conference Proceedings, pp. 119-130, June 1985.
- [14] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, H. Siegel, D. C. Steere "*Coda: A highly available file system for a distributed workstation environment*", IEEE Trans. Computers, Vol. 39, No. 4, Apr. 1990, pp 447-459.
- [15] A. Shamir, "*How to share a secret*", Comm. ACM v.24 n. 11, November 1979.
- [16] D. Stinson, *Cryptography: Theory and Practice*, CRC Press.
- [17] J. G. Steiner, C. Neuman, J. I. Schiller, "*Kerberos, an authentication service for Open Network Systems*", USENIX Association Conferences Proceedings, pp. 191-202, February 1988.
- [18] E. Zadok, I. Badulescu, A. Shender, "*Cryptfs: A stackable vnode level encryption file system*", 1998
- [19] E. Zadok "*FiST: A File System Component Compiler*", PhD Thesis, published as Technical Report CUCS-033-97 (PhD Thesis proposal), Computer Science Department, Columbia University, April 1997.

Design and Implementation of the X Rendering Extension

Keith Packard
XFree86 Core Team, SuSE Inc.
keithp@keithp.com

Abstract

The 2000 Usenix Technical Conference included a presentation outlining the state of the X rendering environment and the capabilities necessary to bring X into the modern world. During the past year, a new extension has been designed and implemented as part of the standard XFree86 distribution.

The X Rendering Extension addresses many of the shortcomings inherent in the core X rendering architecture without adding significantly to the protocol interpretation or implementation burden within the server. By borrowing fundamental image compositing notions from the Plan 9 window system and providing sophisticated and extensible font rendering, XFree86 is now much more able to support existing applications while encouraging new developments in user interfaces. More work remains to be done in areas where best practice is less well established, including precise polygon rasterization and image transformations.

1 Introduction

At the 2000 Usenix conference, the author presented a paper [Pac00a] which discussed the problems inherent in the core X rendering architecture along with some proposals on what a solution might look like. The fundamental problem was that the rendering system had codified practice that was, in hindsight, soon to be obsolete. A rush to release the original X11 standard left no time for research that could have resolved some technical problems. Perhaps the most important issue was that the hardware of that era was not fast enough for the window system to provide a more sophisticated model to interactive applications.

Some of the proposals in last year's paper—alpha compositing, anti-aliasing, sub-pixel positioning and trapezoids—are included in this new extension. Other

parts of the extension, most notably the client-side font management, have been developed during the design of the extension.

The development of this extension has occurred in an entirely open fashion, with input solicited from all areas of the window system community. People involved with XFree86, KDE, Qt, Gdk, Gnome and OpenGL all contributed to the final architecture.

Some areas of the specification are still not complete and the implementation is under construction, but Render has already become an important part of the XFree86 distribution as toolkit and application development starts to take advantage of its presence. It has also delivered a strong message that XFree86 is ready and able to carry the development of the X Window System forward into the future.

2 Rendering Model

The X Rendering Extension (Render) [Pac00b] diverges from the core X [SG92] rendering system by replacing the pixel-value based model with an RGB model. While pixel values are still visible to the client, every pixel value, even those stored in pixmaps, has an associated color value. This provides a natural color-based imaging base while still allowing applications to see the pixel values when necessary.

One compromise resulting from this change involves pseudo-color visuals. The best practice would be to dynamically allocate colors to most closely match the displayed colors. However, the number of pseudo-color desktops is dwindling, so a static color model is used instead. This significantly reduces the implementation burden while still allowing applications to run on pseudo-color hardware. (In fact, the current XFree86 implementation is without even this simplified support and yet no bug reports have been filed to date.)

Along with the presentation of image data as color values, Render supplants the raster-op manipulations of the core protocol with the image compositing operators formally defined by Porter and Duff in 1984 [PD84]. These operators manipulate color data in a natural way by introducing transparency and allowing color data to mix as images are rendered atop one another.

The Porter-Duff compositing model unifies the usual notion of translucency, where a pixel is entirely covered with a non-opaque value, with the notion of partial coverage, where a portion of the pixel is covered while the remaining portion is uncovered. Render uses partial coverage to approximate anti-aliasing; partially covered pixels along the edge of geometric objects are rendered as if they were translucent.

All of the operations in Render are specified in terms of the primitive compositing operators, yielding a consistent model and allowing a minimal implementation. The rendering model is designed to work well with modern toolkits and applications by providing necessary server-resident operations in as simple a fashion as possible.

2.1 Image Compositing

Physically, translucent objects absorb some, but not all, of the light passing through them. The color of the object affects which wavelengths are most strongly absorbed. Visually, translucent objects appear to affect the color and brightness of objects beyond them.

The effect of an opaque object partially obscuring the field of view is similar; at fine enough resolution, the color sampled at a point near the edge of the object will appear as a mixture of the overlying and underlying object colors. Porter and Duff used this property to translate partial coverage into translucency.

The effect of light on a translucent object can be simulated by blending the color of the translucent object with that of objects beyond it. When dealing with computer images, translucency can be described as a mathematical operation on the color data of a collection of images. The Porter and Duff compositing model consists of formulae which use color data in conjunction with a per-pixel opacity value called "alpha". With these formulae many intuitive image manipulations can be performed.

2.1.1 Image Compositing Operators

Each of the operators defined by Porter and Duff operate independently on each of the color channels in each pixel. The equations are abbreviated to show the operation on a single channel of a single pixel.

A common compositing operation is to place one image over another. Transparent areas of the overlying image allow the underlying image to show through. Opaque areas hide the underlying image while translucent areas blend the two images together. By defining the "alpha" of a pixel as a number from 0 to 1 measuring its opacity, a simple equation combines two pixel colors together:

$$C_{result} = C_{under} \cdot (1 - \alpha_{over}) + C_{over} \cdot \alpha_{over}$$

Porter and Duff call this the "over" operator.

Another common operation is to mask an image with another; transparent areas in the mask are removed from the image while opaque areas of the mask leave the image visible.

$$C_{result} = \alpha_{mask} \cdot C_{image}$$

This is the "in" operator. They provide a complete compositing algebra including other operations; only these two are needed for this extension.

One important aspect of this model is that it creates a new image description which attaches another value "alpha" to each pixel. This value measures the "opacity" of the pixel and can be operated on by the rendering functions along with the color components.

2.1.2 Destination Alpha

Sometimes it is useful to create composite images which are themselves translucent, in other words, contain alpha values. This effect can be achieved by augmenting the operators with an operation which produces a composite alpha value along with the color values. For the "over" operator, the composite alpha value is defined as:

$$\alpha_{result} = \alpha_{under} \cdot (1 - \alpha_{over}) + \alpha_{over}$$

The "in" operator composite alpha value is:

$$\alpha_{result} = \alpha_{mask} \cdot \alpha_{image}$$

The resulting images can now be used in additional rendering operations.

2.1.3 Premultiplied Alpha

Visible in the above equations for computing the “over” operator is the asymmetry in the computation of alpha and the color components:

$$\alpha_{result} = \alpha_{under} \cdot (1 - \alpha_{over}) + \alpha_{over}$$

$$C_{result} = C_{under} \cdot (1 - \alpha_{over}) + C_{over} \cdot \alpha_{over}$$

This is “fixed” by respecifying the image data as being “premultiplied by alpha”. Each color component in the image is replaced by that component multiplied by the associated alpha value. Blinn [Bli94] notes that premultiplied images easily provide the correct results when run through long sequences of operations, while non-premultiplied images involve awkward computations. The result is that all four components are now computed uniformly with a single equation:

$$C_{result} = C_{under} \cdot (1 - \alpha_o) + C_{over}$$

2.2 Render Compositing Primitive

The Plan 9 Window System, designed by Russ Cox and Rob Pike [Pik00], provides a unified rendering operation based Porter-Duff compositing:

$$C_{result} = (C_{image} \text{ IN } C_{mask}) \text{ OVER } C_{result}$$

All pixel manipulations are done through this operation which provides a simple and consistent model throughout the rendering system. Render adopts this operation but extends it slightly. Where Plan 9 provides only an OVER operator, Render allows any of the operators defined by Porter and Duff along with a special operator designed for drawing anti-aliased graphics adapted from OpenGL. An illustration of the Render operator is seen in Figure 1.

Using this basic rendering primitive, the extension defines geometric operations by specifying the construction of an implicit mask which is then used in the general primitive above. Anti-aliased graphics can be simulated by generating implicit masks with partial opacity along the edges.

3 Fundamental Render Objects

As with most X extensions, Render adds a number of X-server resident datatypes to encapsulate the notions expressed above:

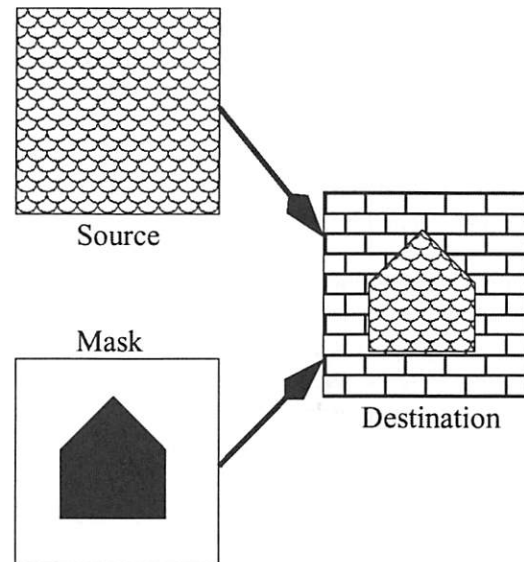


Figure 1: Operation of the compositing primitive

3.1 PictFormat

PictFormats hold information needed to translate pixel values into red, green, blue and alpha channels. The server has a list of picture formats corresponding to the various visuals on the screen along with additional formats that represent data in various formats stored in pixmaps. There are two classes of formats, Indexed and Direct. Indexed PictFormats hold a list of pixel values and RGBA values while Direct PictFormats hold bit masks for each of R, G, B and A.

Direct PictFormats may contain all of R, G, B and A or they may contain only R, G and B or only A. These latter two provide the necessary formats for separate alpha masks and for server visuals which have no destination alpha channel.

Each Indexed PictFormat has an associated colormap from which the associated color values are allocated. This allows multiple different Indexed formats to coexist by allowing applications to select the best matching format and selecting the associated colormap for windows rendered in that format.

3.2 Picture

Pictures connect an X Drawable (Window or Pixmap) with a suitable PictFormat. They also serve as a con-

venient place to place rendering state that is related to the picture. When the `PictFormat` does not provide an alpha channel, the `Picture` may refer to an external alpha channel which is represented as another `Picture`, of which only the alpha channel is used. Without this external alpha channel, the `Picture` has an implicit alpha value of 1 for each pixel.

Pictures are the universal pixel data representation within `Render`. There are no explicit pixel values provided to any operation. To provide for solid colors or repeating patterns, Pictures have a 'Repeat' attribute, when set, the picture is treated as an infinite source of data by tiling the contents of the picture along both axes.

This allows solid color filling, tiling and stippling to be a special case of object-to-object data copying.

One additional property allows for the optimization of image presentation on displays with known sub-pixel geometry. In such environments, applications need control over the compositing of each color component. The usual compositing operator blends all four components using the same alpha value. When the `mask` picture operand in the compositing primitive has the 'ComponentAlpha' attribute set, the R, G, B and A values are interpreted as alpha values operating on each channel in isolation.

3.3 The Composite Request

At the heart of the `Render` extension lies a single request: all other rendering is defined in terms of the `Composite` request which performs the basic composite rendering operation described in Section 2.2. This operator is defined in the `Render` specification as follows:

```
Composite
op: OP
src: Picture
mask: Picture (or None)
dst: Picture
src-x, src-y: Int16
mask-x, mask-y: Int16
dst-x, dst-y: Int16
width, height: Card16
```

This request combines the specified rectangle of `src` and `mask` with the specified rectangle of `dst` using `op` as the compositing operator. The coordinates are relative to their respective drawable's origin. Rendering is clipped to the

geometry of the `dst` drawable and then to the `dst` clip-list, the `src` clip-list and the `mask` clip-list.

If the specified rectangle extends beyond `src`, then if `src` has the `repeat` attribute set, the `src` picture will be tiled to fill the specified rectangle. Otherwise rendering is clipped to the `src` geometry.

If the specified rectangle extends beyond `mask`, then if `mask` has the `repeat` attribute set, the `mask` picture will be tiled to fill the specified rectangle, otherwise rendering is clipped to the `mask` geometry.

If `src`, `mask` and `dst` are not in the same format, and one of their formats can hold all without loss of precision, they are converted to that format. Alternatively, the server will convert each operand to the fallback format.

If `mask` is `None`, it is replaced by a constant alpha value of 1.

When `dst` has `clip-notify` set, a `NoExpose` event is sent if the rendering operation was not clipped by either `src` or `mask`, otherwise a sequence of `GraphicsExpose` events are sent covering areas in `dst` where rendering was clipped by `src` or `mask`.

Some important notes on this definition:

- The operand formats need not match; `Render` automatically converts formats to either the most precise of the provided formats or a fallback internal format in cases where none of the provided formats can hold the data without loss.
- Solid fills, patterns and image copy are all managed by manipulating the 'repeat' attribute of the source `Picture`.
- Geometric objects are drawn by filling 'mask' with an appropriate image, these objects can then be used to stencil any pattern.

3.4 Client-Provided Immediate Data

As `Pictures` provide the only representation for pixel data within `Render`, application generated images must use the existing core `X PutImage` request to transmit that information to the server. A future extension could provide new image transfer functions that would eliminate

the intermediate buffer as well as offer standard image compression algorithms to reduce bandwidth consumed by bulk image data in a networked environment.

4 Text Rendering

Font management and text rendering has always been a source of frustration from both implementors and application developers. X attempted to abstract fonts into simple bitmap images along with associated data described either as simple strings or integer values. Rasterization of the images was left up to the X server and application access to advanced font information, such as kerning tables and ligatures, was not possible through the standard X interfaces.

Many applications, when faced with the X text model simply gave up and implemented all text rendering entirely within the application, sending the resulting rendered images to the X server. The X text rendering code was relegated to drawing labels for buttons and dialog messages. This is not a very efficient use of the tremendous acceleration potential of most graphics systems.

While struggling with building a similar system for the Render extension, several factors converged to redirect development in an entirely new direction. The first was a realization that applications would need direct access to complete font information, preferably the raw font file itself. Only with such direct access could applications be assured that all of the information about the font would be available to them.

An initial Render proposal provided this access by extending the existing X Font Services protocol. Applications and the X server would share font data via the X font server with applications requesting advanced font properties with complex new requests. Taking existing font file formats and generalizing the information so that a single format could encapsulate all information was a daunting task that I put off while developing the simple image composition parts of the extension.

The second factor which changed the direction of the text system was a discussion about PDF files and embedded font data. As with other applications, the display of PDF files requires the presentation of fonts available only to the application. The only portable mechanism for using application provided fonts within the existing font framework is to have the application use the X Font Services protocol by creating a custom font server within

the application. This adds another point of failure for all X applications as they now become unwittingly dependent on this font server whenever they access fonts; adding a large number of such applications will significantly reduce the performance for all applications when manipulating font names.

One goal for Render was to solve this problem in a more straightforward fashion. The obvious solution is to have the application build a font from data it provided, sending glyph images through the X protocol stream instead of through the back door with the X Font Services protocol.

The final factor evolved from a discussion about Unicode encoded fonts. Before a single glyph can be drawn, an X client receives geometric information for every glyph in the font, as well as the minimum and maximum values over the entire font. When using outline fonts, the only way this information can be obtained is to first rasterize every single glyph. For a font with 256 glyphs, this is not a tremendous burden. Encodings containing Han glyphs may contain thousands of glyphs, causing some performance concerns. Unicode fonts can potentially contain millions of glyphs. At this point, it becomes impractical to rasterize all of the glyphs and deliver all of this information to the application, especially when only a small fraction of the font is ever likely to be displayed.

Any new text system would need to be designed to allow the incremental rasterization of glyphs. The problem with incrementally rasterizing glyphs within the X server is that applications would need to incrementally request information about the glyphs, which would entail making extra round-trips at the protocol level. Round trips are a serious performance problem in a networked environment, and this performance penalty would be felt at application startup (which is already a sore point with some X applications).

These factors—the delivery of sophisticated font information, client-generated fonts and the need to incrementally rasterize fonts without increasing the number of round trips—lead to a very simple solution. The X Rendering Extension has no font support. Instead, it provides a mechanism for applications to cache glyph images within the server and rasterize a sequence of them. Applications are responsible for locating fonts, rasterizing glyphs and generating geometric information on their own.

This resolves all three problems while reducing the complexity of the extension. Applications have direct access

to the font files, and thus to all of the information contained therein. As all fonts are client-supplied, embedded fonts in PDF documents are handled as efficiently as any other fonts. Finally, there are no round trips for font handling at all. This reduces application startup time, as no requests are made of the X server to list available fonts or query font information. It also reduces typical network traffic, as only the glyphs actually used by the application transit the network connection. The extra traffic consumed by glyph images is more than compensated for by the lack of glyph metric information for glyphs never drawn on the screen.

Measurements of typical application performance, presented in Section 4.3, show marked decreases both in application startup time and network utilization, even when using 8 bits for each pixel in the glyphs.

4.1 Glyph management

The elimination of server-side fonts within Render presents some new challenges. Applications still need a concise and efficient way of rendering a sequence of glyphs along a fixed baseline. Render provides for the storage of multiple Glyphs in groups known as 'GlyphSets'. Each Glyph is essentially a Picture with additional geometric information that describes where the glyph should be drawn relative to the baseline and an offset to the next glyph.

Glyphs are named within GlyphSets by arbitrary 32-bit numbers: there is no presumed encoding. These names are transmitted in 8, 16 or 32 bit encodings: there is no variable length encoding provided.

Within the current XFree86 implementation, identical glyphs share the same storage. This works within a GlyphSet, and among multiple GlyphSets from one or more clients. This eliminates the server-side storage overhead for having clients provide their own glyphs.

The duplicate rendering and transmission of glyphs can be ameliorated by having multiple applications cooperate in the rasterization of glyphs for particular fonts. The author envisions a cooperative shared-memory mechanism where applications in the same address space can work together to build the needed glyphs. Because the names used to refer to the GlyphSets within the X server have a lifetime no greater than the X connection which created the name, the Render protocol allows each client to have its own name for each GlyphSet. The GlyphSet exists as long as any name exists.

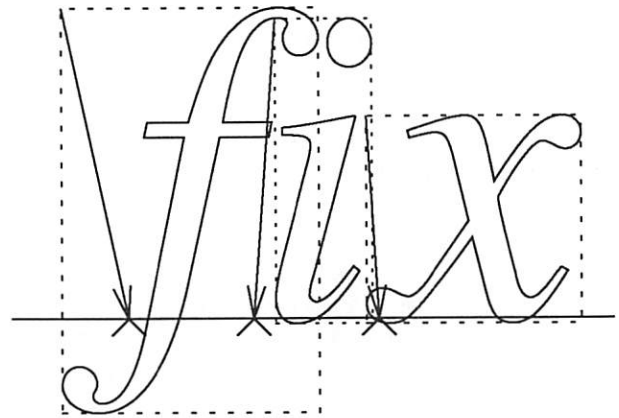


Figure 2: Rendering three glyphs

Significant infrastructure and architecture will need to be designed for this kind of sharing. As these changes will not impact the Render protocol itself, this design can be done when a demonstrated need exists.

4.2 Glyph drawing

Once the needed glyphs are delivered to the X server, the client renders sequences of them with one of the CompositeGlyphs requests (there are three, corresponding to the three glyph name encodings). These requests render a number of glyph lists, each of which is offset from the previous by positional deltas along both axes. Changes to the selected glyph set may be interspersed among the glyph lists. This is very much like the core X text rendering requests with the additional generalization of vertical position adjustments. The position adjustments have also been extended from 8 to 16 bits. An example of simple text rendering is shown in Figure 2. The glyph positions are marked along the baseline with carets while the extent of each glyph image is outlined with a dashed box. Each glyph contains the distance from the upper left corner of the glyph image to the rendering origin, the dimensions of the glyph image and the distance from rendering origin to the location where the next glyph is to be drawn.

When approximating anti-aliasing, a sequence of separate operations using the OVER operator generates inaccurate values when more than one operation covers the same area with alpha values that are neither transparent nor opaque. The problem is that the sub-pixel geometry

Table 1: Network Utilization for Font Data.

	Konqueror	Kword
Number of Lists	29	20
ListFonts	1300 + 113000	888 + 215832
Number of Fonts	21	14
LoadQueryFont	1664 + 41384	1224 + 26900
AddGlyphs	63788 + 0	36840 + 0

of the two objects is lost in the conversion to a coverage value.

The OVER operator assumes that the sub-pixel coverage by two objects is best approximated by assuming that each object covers the same fraction of the other object as of the pixel as a whole. When drawing text, a better approximation is to assume that the glyphs do not actually overlap; the overall area of coverage within the pixel is thus the sum of the areas covered by each pixel.

The glyph drawing requests allow an optional intermediate Picture object to be created; all of the glyphs in the request are rendered to this intermediate Picture using an ADD operator. The resulting image is then rendered to the destination using the operator specified in the request. The server is free to eliminate the intermediate Picture object when the rendering result would not be affected by its use, such as when the glyphs are represented with only a single bit per pixel or when none of the glyphs overlap.

4.3 Network traffic analysis

With glyphs rasterized within the client and transmitted to the X server, there is an obvious concern that these glyphs will represent an additional burden placed on the network. While the glyph images will indeed increase the traffic sent from the client to the server, the traffic in glyph metrics from the server to the client will be eliminated.

It turns out that for typical application execution, the elimination of the glyph metrics and font names transmitted from the server to the client more than compensates for the additional traffic represented by the glyph images. Table 1 shows the network utilization for two common applications using Latin fonts with fewer than 256 glyphs.

Of interest is the large number of bytes needed to simply select appropriate fonts; this is represented by the

ListFonts requests and replies. As the core X architecture provides only primitive string-based font matching, more sophisticated schemes must be implemented within the client, necessitating the transmission of information about available fonts from the server.

Moving from small Latin-1 encoded fonts to larger Han or Unicode fonts will significantly increase the amount of metric data transmitted, while not significantly increasing the amount of glyph data: large parts of the Han or Unicode character set will not be used.

It is important to note that the core X architecture requires a round trip to list or open fonts. As applications typically open many fonts at startup time, these additional round trips can dramatically increase the time it takes to initialize the application.

5 The Xft library

The elimination of font handling within the X server shifts the burden of font file management to the client. Disparate mechanisms for font management among different clients is not (usually) desirable leading to the need for a standard font file access library. Building on the well designed FreeType library, the Xft library provides for common font naming, font file management and font customization. Xft is not a part of the Render extension itself, but is an essential part of the overall architecture for providing font access to applications.

Xft also allows some level of compatibility with older X servers by presenting a unified API that uses the core requests to approximate the results generated with the Render extension. Applications can detect when this happens to allow them to compensate. Fortunately, as XFree86 becomes even more pervasive, the number of legacy X servers should continue to decrease making this compatibility largely unnecessary.

An important premise of the design of Xft was that the library should not hide the underlying rasterization engine and font files from the applications. Any attempt to abstract this access would only serve to prevent applications from taking full advantage of the capabilities present within the font files and rasterization engine.

There is an obvious conflict present here – on one hand, Xft provides enough abstraction to mask the differences between core X fonts and Render based glyphs, and on the other it provides complete access to an underlying

font file if present. Applications will need to prepare for either eventuality and act accordingly. The expectation is that Xft will be used by toolkit libraries, which would be responsible for managing this distinction if necessary.

5.1 Font Names

Selecting fonts is a two part process: first locating an appropriate face and then applying additional attributes that modify the face to create the right glyph images. Once this has been done, attributes about the font are passed back to the application.

Xft unifies these steps into one mechanism. An Xft-FontName is a typed property list; each element has a name and a value. Each available face is represented by an XftFontName containing the properties of the face as provided by the underlying font mechanism.

Applications construct XftFontNames and present them to the API. The library matches this name with the available faces to select the best face and then presents the additional properties to the rasterizer to adjust the final glyph presentation. The resulting font has an associated name which contains additional information about the font, such as the file from which the face was loaded.

While the internal representation of a name is a property list, it is convenient for existing applications to have a string representation which can be converted into the internal representation. The general format for Xft font name strings is:

`< family > - < size > : < name > = < value > ...`

A typical specification might be "times-12" which specifies a 12-point font from the times family. The default values for weight and slant yield a medium-weight roman variant. Even the family and size fields are optional; Xft will choose a suitable family and default size based on the remaining provided attributes. In the minimal case, the font name "" will always match something.

6 Render is Still Under Construction

The pieces of Render described above are not very controversial; they codify existing practice from other systems which is known to work well. They have also been in use for some time, providing some reasonable assurances as to their value. Two further components

of the extension are less well understood and currently not implemented within XFree86. These components, polygon rendering and image transformation, are discussed next, presenting both resolved and outstanding issues with current thinking.

7 Polygon Rendering

Taking cues from OpenGL [SAe99], Render reduces the geometric objects to be rendered by the server to a minimal set. Complex objects are tessellated within the client and sent to the server as a set of primitive objects. This minimizes the implementation effort within the server along with the effort needed to test the conformance of an implementation while not penalizing applications too severely.

Render provides two separate primitive objects; triangles and trapezoids. Both are defined in terms of 32-bit fixed point numbers which use 24 bits for the integer portion of the value and 8 bits for the fractional portion. This allows much more precise location of the vertices for polygons and eliminates a significant source of visual noise caused when objects are snapped to an integer grid.

Triangles are specified by locating their three vertices using these coordinates. Trapezoids are more complex as they are designed to accurately represent the tessellations used by PostScript [Ado85] and Gdtk. Trapezoids are represented by two horizontal lines delimiting the top and bottom of the trapezoid and two additional lines specified by arbitrary points, as shown in Figure 3. Any area between the four lines is a trapezoid (or, in the degenerate case, a triangle). Allowing points not coincident with the top or bottom of the trapezoid makes the edges of multiple trapezoids sharing the same edges align precisely; the same line can be used for all of the trapezoids irrespective of the horizontal extents of those objects.

The rasterization of polygons seems like a non-controversial problem; connect a sequence of vertices with lines and fill the covered area. However, a consequential issue does arise: the specification of an appropriate level of rendering precision.

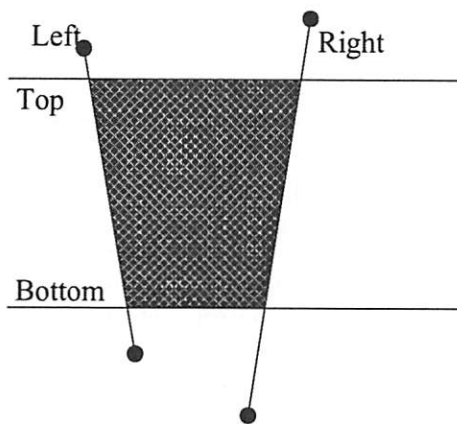


Figure 3: Rendering a Trapezoid

7.1 Precise vs Imprecise

The first question which arises is whether the precise pixelization of polygons should be specified in the standard. The core X protocol requires precise pixelization of all objects, which makes verifying the implementation quite easy, but also essentially eliminated the utility of those objects. The core X specification of most primitives turned out to be too hard to implement efficiently in software, and only some recent hardware has enough flexibility to implement a significant portion in hardware. Very few applications use X geometric objects beyond zero-width lines.

For imprecise rasterization, leaving the pixelization entirely unspecified makes the primitives very difficult to use; applications must accept wide variations in potential presentation. The question is what constraints should be applied to pixelization. OpenGL has relatively weak invariant requirements because of the desire for high-performance mixed software and hardware implementations. Existing applications have stronger requirements for consistent pixelization which require additional constraints.

Precise pixelization places strong requirements on the specification: the pixelization specified must be reasonable to implement as well as reasonable looking. A poor specification can make every implementation useless. A precise specification is also useful for applications that need to mix server side and client side rendering, but only if the specification is straightforward to implement.

Instead of offering only one of these two modes, Render provides both. An imprecise mode designed to map to existing GL-optimized hardware and a precise mode

designed to satisfy the needs of applications requiring detailed control over the rendered results on the screen.

7.2 Imprecise Polygons

Imprecise polygons must match a brief set of invariants:

- *Precise matching of abutting edges.* When specifying two polygons abutting along a common edge, if that edge is specified with the same coordinates in each polygon then the sum of alpha values for pixels inside the union of the two polygons must be precisely one.
- *Translational invariance.* The pixelization of the polygon must be the same when either the polygon or the target drawable are translated by any whole number of pixels in any direction.
- *Sharp edges.* When the polygon is rasterized with Sharp (non anti-aliased) edges, the implicit alpha mask will contain only 1 or 0 for each pixel.
- *Order independence.* Two identical polygons specified with vertices in different orders must generate identical results.

These constraints are designed to minimize the visual artifacts associated with polygon tessellation and translation. It is believed that these invariants can be satisfied with existing hardware.

7.3 Precise Polygons

Precise polygons present a difficult challenge. For sharp polygons, the specification is straightforward: pixels whose center points fall within the polygon are drawn, those outside are not. Following the X model, pixels whose center lie precisely on an edge are drawn when the polygon interior is to the right, or if on a horizontal edge when the polygon interior is below.

Given a 32-bit coordinate space, this can be implemented exactly using values no larger than 64 bits, and that only for clipping.

For smooth (anti-aliased) polygons, the answer is less certain. The obvious answer of computing the fraction of each pixel covered by the polygon turns out to be

computationally expensive; no fewer than 192 bits appear to be required to precisely compute the area covered by a pixel intersected by both sides of the trapezoid.

While this may be better than the computation needed to render wide ellipses from the core protocol, there is no reason to believe that this particular specification is any better than one less expensive to implement. The area covered by a pixel is only a rough approximation to the correct value needed to filter the polygon shape to a collection of pixels; a cheaper specification will be just as "correct", as long as it generates essentially the same values.

A current proposal is to split the pixels involved into three groups: pixels entirely covered by the polygon, pixels entirely uncovered by the polygon and pixels partially covered by the polygon. Covered and uncovered pixels generate the obvious results.

For each pixel partially covered by the trapezoid, the coverage is computed by clipping the trapezoid to the pixel boundaries. Where the trapezoid edges intersect the boundaries of the pixel, the coordinates are represented as 16-bit fractional values along the pixel edge. This precision yields pixel coverage errors of less than 1 part in 2^{15} while requiring only 32 bit arithmetic for the area computation within the pixel.

Another proposal would be to eliminate precise polygons from the extension, leaving only imprecise polygons. Questions remain about what additional invariants would need to be added to the existing list and whether they would impact hardware acceleration for imprecise polygons.

Which choice makes the most sense may well depend on whether an efficient implementation can be written which captures either the above definition of precise polygons or an alternative precise specification. Lacking an efficient implementation, applications will rapidly gravitate to imprecise polygons, leaving precise polygons as yet another albatross within the X server.¹

7.4 Polygon Requests

There is a single request that renders a set of trapezoids and three requests for rendering triangles. Trapezoids are specified by four bounding lines, a top and bottom horizontal line and two diagonal lines, on the left and right. The trapezoid in Figure 3 shows that the coor-

¹ Along with PEX, XIE, LBX, wide lines, ...

ordinates specifying the left and right edges need not be coincident with the horizontal elements.

The triangle requests differ only in the encoding of the triangle vertices. The request formats are taken from OpenGL APIs. The first form delivers a simple list of triangles with one point per vertex in each triangle. The second uses a list of vertices and combines the last two vertices of a triangle with the next vertex to form another triangle. This operation precedes until the list of vertices is exhausted. The final triangle form combines the first and third vertex of a triangle with the next vertex to form the succeeding triangle.

These requests each operate as implicit mask elements in the base Composite operator. A source Picture provides RGBA elements. Additional requests that provide RGBA values for each vertex replace the source Picture with an implicit Picture generated by interpolating the RGBA vertex colors through the polygon. This allows a wide range of color effects with only a few requests.

8 Image Transformation

The final operation added to Render involves the transformation of image data within the X server. Arbitrary affine transformations provide a wealth of possible manipulations that can be accelerated with hardware traditionally used for 3D texture mapping.

The Transform request takes a quadrilateral area from the source image and maps it to a quadrilateral area within the destination area. Vertices are mapped sequentially which allows an arbitrary affine transformation of the image data from source to dest.

The destination quadrilateral forms an implicit alpha mask which may be used to smooth the edges of the transformed image. The source image is created by filtering the source Picture during the transformation. The precise set of filters to be provided has not yet been determined; the expectation is that common hardware filters should be included along with a few higher quality filters designed with digital signal processing techniques.

The eventual intent is to allow implementations to provide additional filters as needed and to create a mechanism within the protocol to advertise at least some of their characteristics.

There are additional questions about edge effects within the filter execution; perhaps additional filter parameters will be needed to generate pixel values beyond the bounds of the source image.

There is also a proposal to limit the destination to a trapezoid rather than the more general quadrilateral form. This would probably simplify the initial implementation while not overly restricting future optimizations.

9 History and Status

The need for the Render extension has been present ever since the X server moved from monochrome to color; the original rendering architecture was never well suited to dealing with color data. However, only with the recent renaissance of X-based application development and consequent reinvigoration of X technology has enough interest and thought been applied to researching what was needed.

Too much weight had been historically given to compatibility with existing X applications and X servers. The two new open source user interface environments, Gnome and KDE, were hamstrung by the existing X rendering system. KDE accepted the limitations of the environment and made the best of them. Gnome replaced server-side rendering with client-side rendering turning the X protocol into a simple image transport system. The lack of hardware acceleration and the destruction of reasonable remote application performance demonstrated that this direction should be supplanted with something providing a modicum of server-side support.

As of Usenix 2000, no formal proposals for a complete extension had been produced and yet considerable interest attended the presentation of a collection of ideas related to this issue. One of the attendees, Rob Pike, described the architecture of the rendering system that he and Russ Cox had developed for the Plan 9 window system. The simple unified architecture from that environment was lifted with only slight extensions to form the core of this new X-based rendering system.

The Render extension protocol was discussed within the XFree86 community for several months. Once it had stabilized, an implementation was started with the goal of producing a workable demonstration of anti-aliased text by August of 2000.

At this point, the implementation provides support only for the basic compositing primitive along with the text primitives. The issues discussed above related to anti-aliased polygon rasterization preclude an implementation of either polygon or image transformation operators. Once that issue has been resolved, the implementation can be completed.

Starting in October of last year, an architecture for accelerating the Render extension has been under development within the XFree86 server. As the protocol has been designed for implementation on modern hardware, the implementation of the primitives themselves has been relatively straightforward. As expected, hardware acceleration provides a tremendous performance benefit. Early measurements of simple image compositing by the author and Mark Vojkovich showed the hardware running as much as forty times faster than reasonably optimized C code.

Late in 2000, Xft support was been integrated into the Qt toolkit, which forms the underpinnings for the K desktop environment. That toolkit provides a complete abstraction for all rendering operations, so the act of modifying the toolkit instantly provided anti-aliased text in all KDE applications. With Qt 3.0, additional Render functionality will be utilized, allowing applications to composite images on the screen.

Some attempts have also been made to utilize Render within the Gnome community. However, until the transition from Gtk+ 1.2 to Gtk+ 2.0, too much of the underlying X font model is exposed to applications to enable a complete transition. Gtk+ 2.0 should be ready within the next year, providing the community with another toolkit free of core X font dependencies.

10 Conclusion

The X Rendering Extension provides a completely new rendering model for use within the X window system. Its small size and low level primitives permit a relatively modest size implementation while providing complete functionality. The primitives have been designed to closely match both application requirements and hardware capabilities.

The X desktop has already begun a transformation with the introduction of anti-aliased text in several toolkits and application suites. Where toolkits were once struggling to provide modern user interface techniques, Ren-

der steps in and permits applications to speak their own language. New applications have been threatening to turn X into a simple image transport protocol; the core rendering system has proven essentially unworkable in the modern world. Render brings graphics back to the server, exposing the capabilities of the hardware while permitting applications to again run efficiently across a network. Render allows the X window system to again support the advancement of the open source desktop environment.

Acknowledgments

The Render extension has been the work of many people, among them:

- Thomas Porter and Tom Duff, who formalized the image compositing operators.
- Rob Pike and Russ Cox, who designed the Plan 9 window system from which the compositing model was lifted.
- Juliusz Chroboczek and Raph Levien, whose proposal for client-side glyph management eliminated font handling from the X server.
- Jon Leech, Brad Grantham and Allen Akin, who patiently explained how OpenGL works.
- Mark Vojkovich, who described how modern 2D graphics hardware functions and for designing an acceleration architecture for Render.
- Dirk Hohndel, who provided the initial spark that touched off the whole thing.
- SuSE, which funds the author's involvement with XFree86.

Thanks also go to Bart Massey for help in preparing the manuscript.

References

- [Ado85] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, 1985.
- [Bli94] Jim Blinn. Compositing theory. *IEEE Computer Graphics and Applications*, September 1994. Republished in [Bli98].
- [Bli98] Jim Blinn. *Jim Blinn's Corner: Dirty Pixels*. Morgan Kaufmann, 1998.
- [Pac00a] Keith Packard. A New Rendering Model for X. In *FREENIX Track, 2000 Usenix Annual Technical Conference*, pages 279–284, San Diego, CA, June 2000. USENIX.
- [Pac00b] Keith Packard. The X Rendering Extension. Xfree86 draft standard, The XFree86 Project, Inc., 2000.
- [PD84] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.
- [Pik00] Rob Pike. *draw - screen graphics*. Bell Laboratories, 2000. Plan 9 Manual Page Entry.
- [SAe99] Mark Segal, Kurt Akeley, and Jon Leach (ed). *The OpenGL Graphics System: A Specification*. SGI, 1999.
- [SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

Scwm: An Extensible Constraint-Enabled Window Manager

Greg J. Badros
InfoSpace, Inc.
2801 Alaskan Way, Suite 200
Seattle, WA 98121, USA
greg.badros@infospace.com

Jeffrey Nichols
School of Computer Science, HCI Institute
Carnegie Mellon University, 5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jeffreyn@cs.cmu.edu

Alan Borning
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA
borning@cs.washington.edu

ABSTRACT

We desired a platform for researching advanced window layout paradigms including the use of constraints. Typical window management systems are written entirely in C or C++, complicating extensibility and programmability. Because no existing window manager was well-suited to our goal, we developed the SCWM window manager. In SCWM, only the core window-management primitives are written in C while the rest of the package is implemented in its Guile/Scheme extension language. This architecture, first seen in Emacs, enables programming substantial new features in Scheme and provides a solid infrastructure for constraint-based window layout research and other advanced capabilities such as voice recognition. We have used SCWM to implement an interface to the Cassowary constraint solving toolkit to permit end users to declaratively specify relationships among window positions and sizes. The window manager dynamically maintains those constraints and lets users view and modify them. SCWM succeeds in providing an excellent implementation framework for our research and is practical enough that we rely on it everyday.

KEYWORDS: constraints, Cassowary toolkit, Scheme, SCWM, X/11 Window Manager

INTRODUCTION

We desired a platform for researching advanced window layout paradigms including the use of constraints. Typical window management applications for the X windows system are written entirely in a low-level systems language such as C or C++. Because the X windows libraries have a native C interface, using C is justified. However, a low-level language is far from ideal when prototyping implementations of sophisticated window manager functionality. For our purposes, a higher-level language is much more appropriate, powerful, and satisfying.

Using C to implement a highly-interactive application also complicates extensibility and customizability. To add a new feature, the user likely must write C code, recompile, relink,

and restart the application before changes are finally available for testing and use. This development cycle is especially problematic for software such as a window manager that generally is expected to run for weeks at a time. Additionally, maintaining all the features that any user desires would result in terrible code bloat.

An increasingly popular solution to these problems is the use of a scripting language on top of a core system that defines new domain-specific primitives. A prime example of this architecture is Richard Stallman's GNU Emacs text editor [40]. In the twenty years since the introduction of Emacs, numerous extensible scripting languages have evolved including Tcl [34], Python [22], Perl [42], and Guile [12, 37]. Each of the first three languages was designed from scratch with scripting in mind. In contrast, Guile—the GNU Ubiquitous Intelligent Language for Extension—takes a pre-existing language, Scheme, and adapts it for use as an extension language.

We are exploring constraint-based window layout paradigms and their user interfaces. Because we are most interested in practical use of constraints, we decided to target the X windows system and build a complete window manager for X/11. We chose to use Guile/Scheme as the extension language for our project that we named SCWM—the Scheme Constraints Window Manager. The most notable feature of SCWM is constraint-based layout. Whereas typical window management systems use only direct manipulation [38] of windows, SCWM also supports a user-interface for specifying constraints among windows that it then maintains using our Cassowary Constraint solving toolkit [1]. Much of the advanced functionality of SCWM is implemented in Scheme, thus exploiting the embedded-extension-language architecture.

BACKGROUND

SCWM leverages numerous existing technologies to provide its infrastructure and support its advanced capabilities.

X Windows and fvwm2

A fundamental design decision for the X windows system [33] was to permit an arbitrary user-level program to manage the various application windows. This open architecture permits great flexibility in the way windows look and behave.

X window managers are complex applications. They are responsible for decorating top-level application windows (e.g., drawing labelled titlebars), permitting resizing and moving of windows, iconifying, tiling, cascading windows, and much more. Many Xlib library functions wrapping the X protocol are specific to the special needs of window managers. Because our goal is to do interesting research beyond that of modern window managers, we used an existing popular window manager, fvwm2, as our starting point [13]. In 1997 when the first author began the SCWM project with Maciej Stachowiak, fvwm2 was arguably the most used window manager in the X windows community. It supports flexible configuration capabilities via a per-user .fvwm2rc file that is loaded once when fvwm2 starts. To tweak parameters, end-users edit their .fvwm2rc files using an ordinary text editor, save the changes, then restart the window manager to activate the changes. The fvwm2 configuration language supports a very restricted form of functional abstraction, but lacks loops and conditionals.

Despite these shortcomings, fvwm2 provides a good amount of control over the look of windows. It also has evolved over the years to meet complex specifications (e.g., the Interclient Communication Conventions Manual [36]) and to deal with innumerable quirks of applications. By our basing SCWM on fvwm2, we leveraged those capabilities and ensured that SCWM was at least as well-behaved as fvwm2. Our fundamental change to fvwm2 was to replace its ad-hoc configuration language with Guile/Scheme [12].

Scheme for Extensibility

Guile [12] is the GNU project's R4RS-compliant Scheme [9] system designed specifically for use as an embedded interpreter. Scheme is a very simple, elegant dialect of the long-popular Lisp programming language. It is easy to learn and provides exceptionally powerful abstraction capabilities including higher-order functions, lexically-scoped closures and a hygienic macro system. Guile extends the standard Scheme language with a module system and numerous wrappers for system libraries (e.g., POSIX file operations).

Embedded Constraint Solver

Cassowary is a constraint solving toolkit that includes support for arbitrary linear equalities and inequalities [1]. Constraints may have varying strengths, and constraint hierarchy theory [6] defines what constitutes a correct solution. We implemented the Cassowary toolkit in C++, Java, and Smalltalk, and created a wrapper of the C++ implementation for Guile/Scheme. Thus, it is straightforward to use the constraint solver in a broad range of target applications.

In addition, the Cassowary toolkit permits numerous hooks for extension. Each constraint variable has an optional attached object, and the constraint solver can be instructed to invoke a callback upon changing the value assigned to any variable and also upon completion of the re-solve phase (i.e., after all variable assignments are completed). SCWM exploits these facilities to isolate the impact of the constraint solver on existing code.

CONSTRAINTS FOR LAYOUT

Ordinary window managers permit only direct-manipulation as a means of laying out their windows. Although this technique is useful, a constraint-based approach provides a more dynamic and expressive system. In SCWM, we use the Cassowary constraint solving toolkit. On top of the primitive equation-solving capabilities of Cassowary, SCWM adds a graphical user interface that employs an object-oriented design. We specify numerous constraint classes representing kinds of constraint relationships, and instances of each class are added to the system for maintaining relationships among actual windows. The interface allows users to create constraint objects, to manage constraint instances, and to create new constraint classes from existing classes by demonstration.

Applying Constraints

Applying constraints to windows is done using a toolbar. Each constraint class in the system is represented by a button on the toolbar (figure 1). The user applies a constraint by clicking a button, then selecting the windows to be constrained. Alternatively, the user can first highlight the windows to be constrained and then click the appropriate button. Icons and tooltips with descriptive text assist the user in understanding what each constraint does. We consulted with a graphic artist on the design of our icons in an effort to make them intuitive and attractive. Preliminary user studies have demonstrated that users can determine the represented relationship reasonably well from the icons even without the supporting tooltip text.

We provide the following constraint classes in our system. Many interesting relationships are either present or can be created by combining classes in the list.

Constant Height/Width Sum Keep the total of the height/width of two windows constant.

Horizontal/Vertical Separation Keep one window always to the left of or above another.

Strict Relative Position Maintain the relative positions of two windows.

Vertical/Horizontal Maximum Size Keep height/width of a window below a threshold.

Vertical/Horizontal Minimum Size Keep height/width of a window above a threshold.

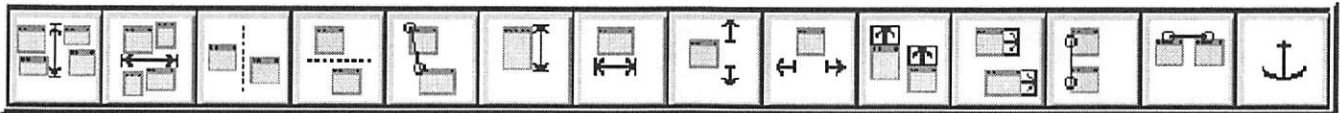


Figure 1: Our constraint toolbar. The text describes the constraint classes in the same order as they are laid out in the toolbar (from left to right).

Vertical/Horizontal Relative Size Keep the change in heights/widths of two windows constant (i.e., resize them by the same amount, together).

Vertical/Horizontal Alignment Align the edge or center of one window along a vertical/horizontal line with the edge or center of another window.

Anchor Keep a window in place.

Some of these constraint types can constrain windows in several different ways. For example, the “Vertical Alignment” constraint can align the left edge of one window with the right edge of another or the right edge of one window with the middle of another. Users specify the parameters of the relationship by using window “nonants,” the ninefold analogue of quadrants (figure 2). The nonant that the user clicks in dictates the part of the window to which the constraint applies. For example, if the user selects the “Vertical Alignment” constraint and chooses the first window by clicking in any of the east nonants, and the second window by clicking on its left edge, the resulting constraint will align the right edge of the first window with the left edge of the second. This technique makes some constraint classes, such as alignment, more generally useful. It also decreases the number of buttons on the toolbar, which could otherwise become unwieldy with many narrowly-applicable constraint classes.

NW 0	N 1	NE 3
W 3	C 4	E 5
SW 6	S 7	SE 8

Figure 2: The nine nonants of a window.

Managing Constraints

Once a constraint is applied, the user still needs to be able to manage it. Users may wish to disable the constraint temporarily or remove it entirely. They may encounter an odd behavior while they are moving or resizing a window and want to discover which constraint(s) caused the unexpected result, they may simply be curious to know what constraints are applied to a given window and how that window will interact with other windows. Our constraint investigation interface allows for all of these kinds of interactions.

The constraint investigation window allows the user to enable or disable constraints using checkboxes, and to remove constraints using a delete button. The window is dynamically updated as constraints are applied and removed, and changes made in the investigator are immediately reflected in the layout of windows.

When the user moves her mouse pointer over a constraint in the investigator, the representation of that constraint is drawn directly on the windows related by the constraint (figure 3). This hint makes it easy for the user to make the correct associations between windows and constraints. Each constraint class defines its own visual representation, which in most cases closely matches the icon in the toolbar.

Enabling or disabling constraints can result in global rearrangements of windows and large changes in position. To make these discontinuities less confusing, we animate windows fluidly from their old positions and sizes to their new configuration. The animations borrow features from the Self programming environment that mimic cartoon-style animation [7].

Constraint abstractions

A problem with the interface as described thus far is that the basic constraint classes, such as “Vertical Alignment” and “Horizontal Separation,” are not always sufficient to convey a user’s intention fully. Our own use showed that often one needs to combine several constraints to obtain the desired behavior. A good example of this situation is tiling (figure 4), where two or more windows are aligned next to each other such that they appear to become a window unit of their own. A tiling configuration for two windows can take from three to five constraints to implement. Adding the constraints is tedious when tiling many windows, or when repeatedly tiling and untiling two windows. Certainly a “tiled windows” constraint class could be hard-coded into the system, but that just postpones the problem—some means of abstracting relationships should be provided to the end user.

Our solution to this problem is to support constraint “compositions.” A composition is created using a simple programming-by-demonstration technique. We record the user applying a constraint arrangement to some windows in the workspace. The constraints used and the relationships created among the windows are saved into a new constraint class object, which then appears in the toolbar like all other constraint classes. Clicking the button in the toolbar will prompt the user to select a number of windows equal to that used in the recording. The constraints will then be

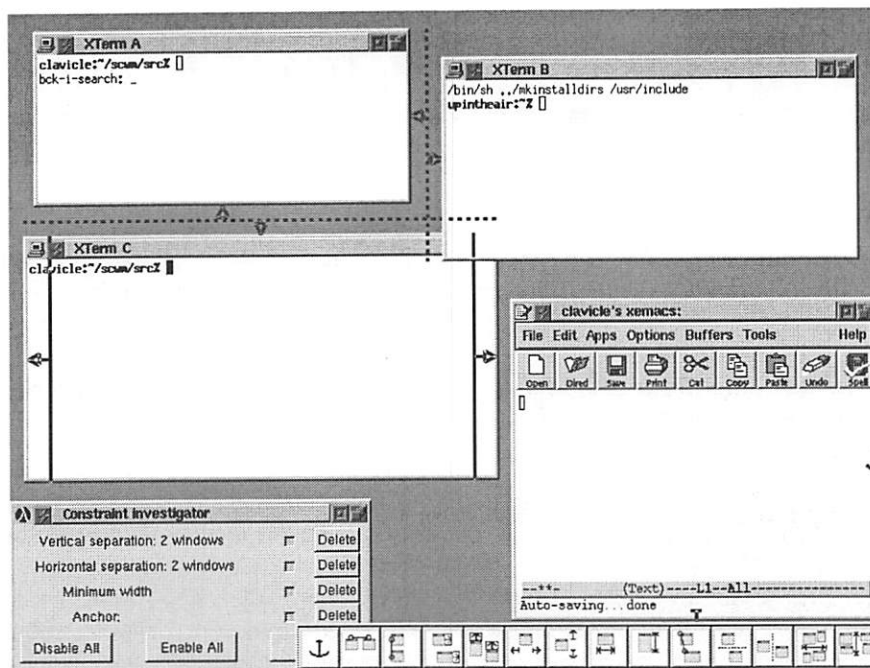


Figure 3: Visual representation of constraints. XTerm A is constrained to be to the left of XTerm B, and above XTerm C. Additionally, XTerm C is required to have a minimum width, and the XEmacs window's southeast corner is anchored at its current location. The constraint investigator that allows users to manage the constraints instances appears in the bottom left of the screen shot.

applied in the same order as before. Compositions allow users to accumulate a collection of often-used constraint configurations that can then be easily applied.

Inferring Constraints

Our toolbar-based user interface allows flexible relationships to be specified, but many common user desires reflect very simple constraints. For example, users may place a window directly adjacent to another window and want them to stay together. Some windowing systems provide a basic “snapping” behaviour that recognizes when a user puts a window nearly exactly adjacent to another window and then adjusts the window coordinates slightly to have them snap together precisely.

In SCWM, we support a useful extension to basic snapping called “augmented snapping” [15]. Using this technique, the user has the option of transforming a snapped-to relationship to a persistent constraint that is then maintained during subsequent manipulations. When a snap is performed, instead of simply moving the window, the appropriate constraint object is created and added to the system. Such inferred constraints can be manipulated via the constraint investigator described earlier. They also can be removed by simply “ripping-apart” the windows by holding down the Meta modifier key while using direct manipulation to move them apart.

USABILITY STUDY

We applied a discount usability approach [32] to improve our constraint interface to managing windows.

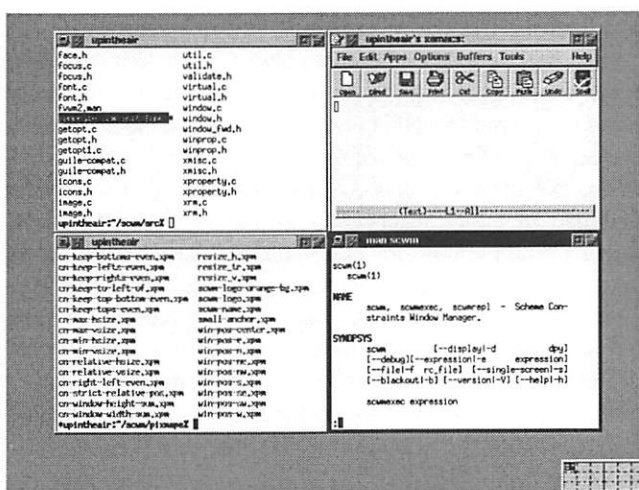


Figure 4: Four windows tiled together. Unlike tiled-only window managers, SCWM permits users to tile a subset of their windows; other windows could overlap arbitrarily.

Methodology

Six advanced computer users thought aloud while performing three tasks. Each task consists of two parts: discovery and re-creation. First, users manipulate windows with constraints already active to discover and describe those relationships (without use of the constraint investigator). After giving a correct description, they then use the interface on a second display to constrain a fresh set of windows identically. Users were given only a very minimal description of the interface.

The three constraint configurations tested were: 1) a Netscape Find dialog kept in the upper right corner of the main browser window; 2) three windows kept right-aligned along the edge of the screen such that none of the windows overlap nor leaves the top or bottom of the screen; and 3) two windows tiled horizontally.

Results

All users were able to complete their tasks. Discovering the constraints was straightforward—manipulating the windows and observing the behaviour was sufficient to deduce the relationships already present. Re-creating the configurations was more troublesome, but users still succeeded. They often used the investigator to remove incorrect constraints, but then continued onward with an alternate hypothesis.

Problems discovered

Our study uncovered numerous usability issues. The most substantial problem involved selecting window parts for the alignment constraints. When performing a vertical alignment, all that matters is whether the user clicks on the left, center, or right third of the window—it is irrelevant whether the click is in the top, middle, or bottom of the window. Our interface, however, still highlighted individual corners or edges as it does for anchor constraints where any of the nine positions is significant. Users were confused by the UI distinguishing along the irrelevant vertical dimension. We revised SCWM to highlight whole edges of windows when applying an alignment constraint.

When users began adding a constraint and wanted to cancel, they were unsure of how to abort their action. Some users clicked on the toolbar thinking that is a special window. Others discovered that clicking on the background results in an error that terminates the operation. No user realized that a right-click aborts and we now also support pressing the `Esc` key to cancel a window selection.

Other observations

The users who performed best studied the tooltip help for each of the toolbar buttons before attempting their first re-creation sub-task. We were surprised at the variety of constraints used in re-creating our configurations: no user matched the expected solution on all three tasks. In particular, the “strict relative position” constraint was used especially advantageously by users who chose to configure windows manually before applying constraints to keep the

```
SCWM_PROC( X_property_get,
           "X-property-get",
           2, 1, 0,
           (SCM win, SCM name, SCM consume_p))
/* Get X property NAME of window WIN. */
#define FUNC_NAME s_X_property_get
{
    SCM answer;
    VALIDARG_WIN_ROOTSYM_OR_NUM_COPY(1, win, w);
    VALIDARG_STRING_COPY(2, name, aprop);
    VALIDARG_BOOL_COPY_USE_F(3, consume_p, del);
    ...
    XGetWindowProperty(...);
    ... answer = ...;
    return answer;
}
#undef FUNC_NAME
```

Figure 5: An example SCWM primitive.

```
(define*-public (window-class
                 #&optional (win (get-window)))
  "Return the class of window WIN."
  (X-property-get win "WM_CLASS"))
```

Figure 6: The “window-class” procedure.

windows as they were.

Not all users discovered the constraint-visualization feature of the investigator. We now draw the visualizations whenever the user points at any part of the description, not just the enable checkbox. Also, one user wanted to modify the parameters of a constraint in the investigator window directly.

THE SYSTEM

SCWM is a complex software system that emphasizes extensibility and customizability to enable sophisticated capabilities to be developed and tested quickly and easily.

The current implementation of SCWM contains roughly 32,500 non-comment, non-blank lines of C code, 800 lines of C++ code, and 25,000 lines of Scheme code. The Guile/Scheme system is about 44,000 lines of C code and 11,500 lines of Scheme code. Finally, the Cassowary constraint solving toolkit is about 9,500 lines of C++ code in its core, plus about 1,400 lines of C++ code in the Guile wrapper. The following subsections describe various technical aspects of the implementation of SCWM in greater detail.

Basic philosophy

Our first version of SCWM was a simple derivative of its predecessor, `fvwm2`, with the ad-hoc configuration language replaced by Guile/Scheme. Like `fvwm2`, SCWM reads a startup file containing all of the commands to initialize the settings of various options. Most `fvwm2` commands have reasonably straightforward translations to SCWM sentential expressions. For example, these `fvwm2` configuration lines:

```

Style "" ForeColor black
Style "" BackColor grey76

HighlightColor white navyblue

AddToFunc Raise-and-Stick
+ "I" Raise
+ "I" Stick

Key s WT CSM Function Raise-and-Stick

```

are rewritten for SCWM in Guile/Scheme as:¹

```

(window-style "" #:fg "black"
              #:bg "grey76")

(set-highlight-foreground! "white")
(set-highlight-background! "navyblue")

(define* (raise-and-stick
          #&optional (win (get-window)))
  (raise-window win)
  (stick win))

(bind-key '(window title) "C-S-M-s"
          raise-and-stick)

```

The simpler and more regular syntax is convenient for the end user. An even greater advantage of using a real programming language instead of a static configuration language stems from the ability to extend the set of commands (either by writing C or Scheme code) and to combine those new procedures arbitrarily.

Adding a new SCWM primitive is easily done by writing a new C function that registers itself with the Guile interpreter. For example, after using C to add the “X-property-get” primitive (figure 5), we can write a new procedure to report a window’s class, which is just the value of its `WM_CLASS` property (figure 6). Then we can use that procedure interactively by writing:

```

(bind-key 'all "C-S-M-f"
  (lambda ()
    (let* ((win (window-with-focus))
           (class (window-class win)))
      (if (string=? class "Emacs")
          (resize-window 500 700 win)
          (resize-window 400 300 win))))))

```

The above expressions, when evaluated in SCWM’s interpreter, will make the user’s “Control + Shift +

¹Because the `fvwm2` configuration language is so limited, it is possible to mechanically convert to SCWM commands; we provide a reasonably-complete automated translator for this purpose.

Meta + f” keystroke resize the window to either 500 × 700 pixels if the currently-focused window is an Emacs application window, or 400 × 300 pixels otherwise.

SCWM’s extensible architecture also allows Guile extensions to be accessible from the window manager. Via standard Guile modules, SCWM can read and parse web pages, download files via ftp, do regular expression matching, and much more. In fact, nearly all of the user-interface elements in SCWM are built using `guile-gtk`, a Guile wrapper of the GTK+ toolkit.

Binary Modules

Because each user only needs a subset of the full functionality that SCWM provides, it is important that users only pay for the features they require (in terms of size of the process image). Guile, unlike Emacs Lisp, allows new primitives to be defined by dynamically-loadable binary modules. Without this feature, all primitives would need to be contained in the SCWM core, thus complicating the source code and increasing the size of the resulting monolithic system.

The voice recognition module based on IBM’s ViaVoice™ software illustrates the benefits of dynamically-loaded extensions. Those users who do not to use that feature—perhaps because the library is not available on their platform or perhaps because they have no audio input device—will never have the module’s code loaded.

Implementing the module was also straightforward. After getting a sample program from IBM’s ViaVoice™ voice recognition engine working, it required less than six hours of development effort to wrap the core functionality of the engine with a Scheme interface. A grammar describes the various utterances that SCWM understands, and the C code asynchronously invokes a Scheme procedure when a phrase is recognized. Because those action procedures are written in Scheme, the responses to phrases can be easily modified and extended without even restarting SCWM.

Graphical configuration

Another example of the extensibility that Guile provides SCWM is the preferences system for graphical customization. Novice SCWM users are unlikely to want to write Scheme code to configure the basic settings of their window manager, such as the background color of the currently-active window’s titlebar. A graphical user interface is necessary to manage these settings, but there are potentially a huge number of configurable parameters. Undisciplined maintenance of a user interface for those options would be tedious and error-prone.

Fortunately, SCWM can leverage its Scheme extension language to ease these difficulties. The `defoption` module provides a macro `define-scwm-option` that permits declarative specification of a configuration option.² To expose a graphical interface to the

²Recent versions of Emacs [40] provide a similar feature in their “cus-

`*highlight-background*` configuration variable, the SCWM developer need simply write:

```
(define-schw-option
  *highlight-background* "navy"
  "The bg color for focused window."
  #:type 'color
  #:group 'face
  #:setter (lambda (v)
             (set-highlight-background! v))
  #:getter (lambda () (highlight-background)))
```

This code states that `*highlight-background*` is an end user configurable variable that will contain a value that is a color. It also specifies that the variable can be grouped with other variables into a `face` category. Finally, setter and getter procedures are specified to teach SCWM how to alter and retrieve the value.

The `preferences` module then accumulates all of these specifications and dynamically generates the user interface shown in figure 7.³ This modular approach also enforces the separation of the visual appearance from the desired functionality—a visually-distinct notebook-style interface with the same functionality is also available.

Connecting to Cassowary

The most important module for our research on advanced window layout paradigms is the wrapper of the Cassowary constraint solving toolkit. To connect the constraint solver with the window manager, the variables known to the solver must relate to aspects of the window layout. Each application window object contains four constrainable variables: `x`, `y`—the offsets of the window from the top-left corner of the virtual desktop; and `width`, `height`—the dimensions of the window frame in pixels. When Cassowary finds a new solution to the set of constraints, it invokes a hook for each constraint variable whose value it changes, and invokes another hook after all changes have been made. For SCWM, the constraint-variable-changed hook adds the window that embeds that constraint variable to its “dirty set,” and the second hook repositions and resizes all of the windows in the dirty set.

In each window object, the constrainable variables that correspond to the window’s position and size mirror the ordinary integer variables that the rest of the application uses. The hooks copy the new values assigned to the constrainable variables into the ordinary variables. This technique avoids modifying the vast majority of the code that manipulates and manages windows. (Bjorn Freeman-Benson discusses these issues in greater detail [11].)

To make it easy for developers to express constraints among “tomize” package. The layout of their user-interfaces is simpler, though, as no attempt is made to create a fully graphical interface.

³The user interface is written in `guile-gtk`, a Guile wrapper of the GTK+ widget toolkit [18] that integrates seamlessly with SCWM.

windows, the constraint variables embedded in each window are available to Scheme code via the accessor primitives `window-clv-{xl,xr,yt,yb,width,height}`, where, for example, `-xl` names the `x` coordinate of the left side of the window and `-yb` abbreviates the `y` coordinate of the bottom of the window.⁴ Thus, to keep the tops of two window objects aligned, we can use:

```
(cl-add-constraint solver
  (make-cl-constraint
    (window-clv-yt win1) =
    (window-clv-yt win2)))
```

RELATED WORK

There is considerable early work on windowing systems [16, 17, 26, 25, 27, 23]. Many of these projects addressed lower-level concerns that a contemporary X/11 window manager can ignore. An issue that does remain is tiled vs. overlapping windows. SCWM, like nearly all windowing interfaces of the 1990s, chooses overlapping windows for their generality and flexibility. However, unlike other systems, SCWM’s constraint solver can permit arbitrary sets of windows to be maintained in a tiled format of a given size.

Although there are literally dozens of modern window managers in common use on the X windowing platform, only two (besides `fvwm2`) are especially related to SCWM. GWM, the Generic Window Manager, embeds a quirky dialect of Lisp called “WOOL” for Window Object Oriented Language [30]. It supports programmability, and some of its packages, such as directional focus changing, inspired similar modules in SCWM. Sawfish [19] is a more recent window manager with an architecture similar to GWM and SCWM. Like GWM, it embeds its own unique dialect of Lisp (called “rep”). Both embrace the extensibility language architecture and provide low level primitives, then implement other features in their extension language. However, the embedded Lisp dialects used by GWM and Sawfish both suffer from the lack of lexical closures that Scheme provides SCWM. Neither GWM nor Sawfish has any constraint capabilities, though the hooks they provide can permit procedural implementations to approximate some of the simpler constraint-based behaviours that SCWM implements.

Various other scripting languages exist. As mentioned previously, GNU Emacs and its Emacs Lisp is similar to SCWM in philosophy. The earliest popular general-purpose scripting languages is Tcl, the tool command language [34]. John Ousterhout, Tcl’s author, makes a compelling case for the advantages of scripting [35]. Tcl is an incredibly simple but under-powered language that only in the most recent versions includes real data structures. Subsequent similar languages include Python [22] and Perl [42]; both are far more feature-full languages than Tcl, but all three are more commonly

⁴For each window, explicit constraints `xr = x + width` and `yb = y + height` are added automatically by SCWM.

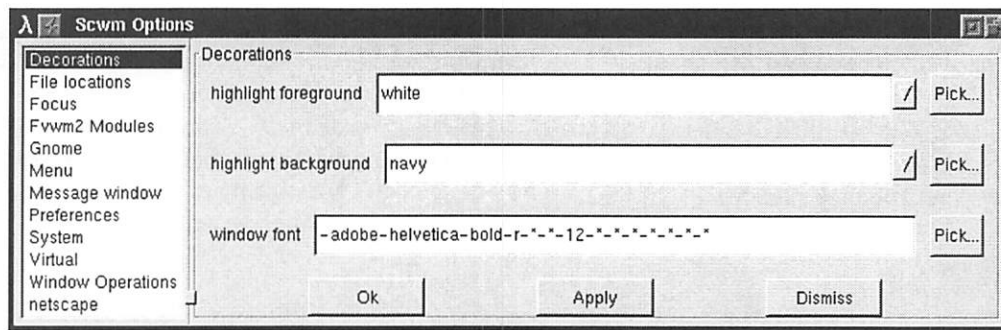


Figure 7: The automatically-generated options dialog.

used for scripting where the main control resides with the language. SCWM and Emacs both exploit their languages for embedding and invoke scripting code in response to events dispatched by C code.

There are also several other Scheme-based extension languages. Elk [10] is an early Scheme intended as an extension language but is no longer well supported. SIOD (Scheme In One Defun) [39] is an especially compact implementation of Scheme that in return compromises completeness and standards-compliance; it is embedded in the popular GIMP (GNU Image Manipulation Program) application [14] to support user-programmable transformations on images.

Numerous other application domains have used constraint solvers. Early work includes the drawing tool Sketchpad [41] and the simulation laboratory ThingLab [5]. Many other drawing programs have embedded constraint solvers over the years including Juno [31], Juno-2 [21], Unidraw [20], and Penguin [8]. Unidraw and Penguin both leverage QOCA, a constraint solver that (like Cassowary) is able to maintain arbitrary linear arithmetic constraints [24]. SCWM includes more than just constraints in its support for intelligent window layout; another paper describes some of its other layout capabilities [3].

Web browser layout presents challenges similar to window layout. Our “Constraint Cascading Style Sheets” work also embeds Cassowary and exposes a declarative specification language to web authors for describing page layout [2]. Widget layout in user interfaces is yet another two-dimensional layout problem. Amulet [29] and the earlier Garnet [28] both provided constraint solvers based on simple local propagation techniques. These solvers suffer from an inability to handle inequalities and simultaneous equations, which unfortunately arise all too often in the natural declarative specification of layout desires.

CONCLUSIONS AND FUTURE WORK

One of the most useful aspects of this research has been the continuous feedback from our end users throughout the development of SCWM. Since 1997, we have made the latest version of SCWM (along with all of its source code) available on the Internet, and have actively solicited feedback on our

support mailing lists. Many of the high-level layout features were developed in response to real-world frustrations and annoyances experienced either by the authors or by our user community. Although cultivating that community has taken time and effort, we feel that the benefits from user feedback outweigh the costs.

Perhaps the most significant implementation issue for SCWM is its startup time of nearly 20 seconds on a Pentium III 450 class machine. Loading the nearly 20,000 lines of Scheme code at every restart is costly, and wasteful. To address this, we should add an Emacs-like “unexecing” capability to dump the state of a SCWM process that has all of the basic modules loaded. Although this would increase the size of the executable, it also would substantially reduce startup delays. Fortunately, after startup, SCWM’s performance is indistinguishable from other window managers that are written entirely in C.

Another rich area for future work involves our constraint interface. Currently, we only support constraints among windows. It seems useful to permit the addition of “guide-line” and “guide-point” elements and allow windows to be constrained relative to them. These could, for example, be used to ensure that a window stays in the current viewport, or stays in a specific region of the display. It would also be intriguing to investigate the possibility of ghost-frame objects that are controlled exclusively by SCWM. These window frames could then hold real application windows by dragging them into the frame. This feature would permit hierarchically organizing windows, while still allowing full access to the constraint solver for non-hierarchical relationships.

We are also considering extending our voice-based interface to permit specifying constraints. In SCWM, a user can center a window simply by saying aloud “Center current window.” The voice recognition interface to window layout and control encourages the user to express higher level intention: it is far more awkward to say “move window to 379, 522” than it is to say “move window next to Emacs.” In this way, the voice interface usefully contrasts with direct manipulation where exact coordinates naturally result from the interaction technique. Additionally, voice-based interactions may prove especially valuable for disabled users for whom direct

manipulation is difficult.

Discerning a user's true intention is an interesting complexity of the declarative specification of our current constraints interface. Consider a user who is manipulating three windows, **A**, **B**, and **C**. Suppose the user constrains **A** to be to the left of **B**, and **B** to the left of **C**. Now suppose the application displaying in window **B** terminates, thus removing that window. Should window **A** still be constrained to be to the left of window **C**? In other words, should the transitive constraint that was implicit through window **B** be preserved? The answer depends on the user's underlying desire. Providing higher-level abstractions for commonly-desired situations may alleviate this ambiguity. For example, if the user had pressed a button to keep three windows horizontally non-overlapping in a row, it is clear that window **B**'s disappearance should not remove the constraint that window **A** remain to the left of **C**.

Finally, we are especially interested in combining our work on constraints and the web [2] with this work on window layout. Web, window, and widget layout are all fundamentally related, and their similarities should ideally be factored out into a unifying framework so that advances made in any area benefit all kinds of flexible, dynamic two-dimensional layout.

ACKNOWLEDGMENTS

We thank Maciej Stachowiak, Sam Steingold, Robert Bihlmeyer, and Todd Larason for their contributions to the SCWM project. Thanks to Craig Kaplan for his helpful comments on a draft of this paper. This research has been funded in part by both a National Science Foundation Graduate Research Fellowship and the University of Washington Computer Science and Engineering Wilma Bradley fellowship for Greg Badros, and in part by NSF Grant No. IIS-9975990.

Availability

SCWM and Cassowary are both freely available on the Internet [4, 1] and are distributed under the terms of the GNU General Public License.

REFERENCES

1. Greg J. Badros and Alan Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998. <http://www.cs.washington.edu/research/constraints/cassowary/cassowary-tr.pdf>.
2. Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, November 1999.
3. Greg J. Badros, Jeffrey Nichols, and Alan Borning. SCWM—an intelligent constraint-enabled window manager. In *Proceedings of the AAAI Spring Symposium on Smart Graphics*, March 2000.
4. Greg J. Badros and Maciej Stachowiak. Scwm—The Scheme Constraints Window Manager. Web page, 1999. <http://scwm.sourceforge.net/>.
5. Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).
6. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992. <http://www.cs.washington.edu/research/constraints/theory/hierarchies-92.html>.
7. Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *Proceedings of the 1993 ACM Conference on User Interface Software and Technology*, pages 45–55, Atlanta, Georgia, November 1993. User Interface Software and Technology.
8. Sitt Senn Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of UIST 1998*, San Francisco, California, November 1998.
9. William Clinger and Jonathan Rees. *Revised 4 Report on the Algorithmic Language Scheme*, November 1991.
10. Elk—the extension language kit. Web page, 1999. <http://www-rn.informatik.uni-bremen.de/software/elk>.
11. Bjorn Freeman-Benson. Converting an existing user interface to use constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.
12. FSF. Guile—The GNU Ubiquitous Intelligent Language for Extension. Web page, 1999. <http://www.gnu.org/software/guile/guile.html>.
13. fvwm—the f? virtual window manager. Web page, 1999. <http://www.fvwm.org>.
14. Gimp—GNU image manipulation program. Web page, 1999. <http://www.gimp.org>.
15. Michael Gleicher. Integrating constraints and direct manipulation. In *Proceeding 1992 Symposium on Interactive 3D*, pages 171–174, 1992.

16. James Gosling. SunDew – a distributed and extensible window system. In *Methodology of Window Management*, chapter 5, pages 47–57. Springer Verlag, Heidelberg, Germany, 1986.
17. James Gosling and David Rosenthal. A window manager for bitmapped displays and unix. In *Methodology of Window Management*, chapter 13, pages 115–128. Springer Verlag, Heidelberg, Germany, 1986.
18. GTK+—the GIMP toolkit. Web page, 1999. <http://www.gtk.org>.
19. John Harper. Sawfish. Web page, 1999–2000. <http://sawmill.sourceforge.net/>.
20. Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. *An Object-Oriented Architecture for Constraint-Based Graphical Editing*, chapter 14, pages 217–238. Springer, 1995.
21. Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California, December 1994.
22. Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.
23. Mark S. Manasse and Greg Nelson. *Trestle Reference Manual*. Digital Systems Research Center, December 1991. <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-068.html>.
24. Kim Marriott, Sitt Sen Chok, and Alan Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming*, 1998.
25. Brad Myers. Issues in window management design and implementation. In *Methodology of Window Management*, chapter 6, pages 59–71. Springer Verlag, Heidelberg, Germany, 1986.
26. Brad A. Myers. The user interface for Sapphire. *IEEE Computer Graphics and Applications*, 4(12):13–23, December 1984.
27. Brad A. Myers. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications*, 8(5):65–84, September 1988.
28. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojchick. The Garnet toolkit reference manuals: Support for highly-interactive graphical user interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.
29. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferreny, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.
30. Colas Nahaboo. GWM—the generic window manager. Web page, 1995. <http://www.inria.fr/koala/gwm>.
31. Greg Nelson. Juno, a constraint-based graphics system. In *Proceedings of SIGGRAPH 1985*, San Francisco, July 1985.
32. Jakob Nielson. *Usability Engineering*. Morgan Kaufmann, 1994.
33. Adrian Nye. *Xlib Programming Manual*. O'Reilly & Associates, Inc., Sebastopol, California, 1992.
34. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.
35. John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.
36. David Rosenthal. *Inter-client Communications Convention Manual*, version 2.0 edition, 1994. <http://www.talisman.org/icccm>.
37. Peter H. Salus, editor. *Functional and Logic Programming Languages*, volume 4 of *Handbook of Programming Languages*, chapter 4. MacMillan Technical Publishing, Indianapolis, Indiana, 1998.
38. Ben Schneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.
39. SIOD—scheme in one defun. Web page, 1999. <http://people.delphi.com/gjc/siod.html>.
40. Richard M. Stallman. EMACS: The extensible, customizable display editor. Technical Report 519a, Massachusetts Institute of Technology Artificial Intelligence Laboratory, March 1981. <http://www.gnu.org/software/emacs/emacs-paper.html>.
41. Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.
42. Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.

The X Resize and Rotate Extension - RandR

Jim Gettys

Cambridge Research Laboratory, Compaq Computer Corporation.

Jim.Gettys@Compaq.com

Keith Packard

XFree86 Core Team, SuSE Inc.

keithp@keithp.com

Abstract

The X Window System protocol, Version 11, was deliberately designed to be extensible, to provide for both anticipated and unanticipated needs. The X11 core did not anticipate that the properties of X server screens might need to change dynamically, as occurs frequently with desktops, laptops and hand held computers not envisioned in the 1980's.

The Resize and Rotate extension (RandR) is a very small set of client and server extensions designed to allow clients to modify the size, accelerated visuals and rotation of an X screen. RandR also has provisions for informing clients when screens have been resized or rotated and it allows clients to discover which visuals have hardware acceleration available.

RandR needs to be discussed in concert with recent developments in X server implementation and the new Render extension to understand the implications of the aggregate. In isolation, RandR seems to provide a limited but useful improvement, but together with the Render extension and reimplementation of the X server rendering code, RandR provides part of a key change in X Window System capabilities. We believe this will enable much easier migration and replication of applications between X servers for pervasive computing. This paper also describes this vision.

1 Introduction

The X Window System [SG92] extension framework has served us well, allowing significant extensions to the X

design over the last 14 years. This extension framework has encouraged new functionality to be introduced, and has encapsulated optional functionality permitting clean non-universal deployment. Probably more importantly, the extension framework has isolated "bad" ideas from the core X functionality allowing their eventual atrophy into irrelevance.¹

At the time X11 was designed, we did not anticipate that the properties of X server screens might need to change dynamically. However, this routinely occurs today with laptops, and handheld computers.

Even most current desktop systems share this need. As there is usually a significant performance and display memory tradeoff between screen resolution and depths, the ability to change display characteristics without restarting your X session is needed by general users (particularly gamers).

Laptops and handheld computers need to change their screen size to drive external monitors at different resolutions than their built in screens. Permitting these portable devices to rotate their display provides for better use of screen real-estate in applications that prefer displays with the rotated aspect ratio. It is convenient to flip a laptop sideways when reading documents intended for paper presentation—the aspect ration more closely matches the document leaving less wasted space. Projectors using mirror systems may find flipped and/or rotated screens very useful. From the applications perspective, rotating the screen is essentially the same as changing the screen size.

We expect that most applications can remain relatively oblivious to screen size changes, though simple modi-

¹ E.g. PEX, XIE, LBX, along with wide lines and arcs in the core protocol ...

fications may be required in toolkits and window managers. The only real challenge which screen size change presents to most applications is in ensuring that menus stay visible on the screen. Menus are an important special case as they are typically the only user interface elements not managed by the window manager. Applications must remain informed about the size of the screen to ensure that menus do not extend beyond the boundaries of the screen making portions inaccessible to the user. As menus are generally provided by toolkits, rather than directly by applications, simple changes within the toolkits should resolve these problems. Toolkits may also wish to take advantage of acceleration information provided by RandR to maximize performance.

1.1 Rendering

The true physical “depth” of a display’s frame buffer may need to change to either support different screen resolutions due to limitations in the size of display memory or for performance. Since applications currently have a static view of visual types available on a server, and changing visuals dynamically would likely break too many applications, the visuals advertised by the X server will not change when frame buffer depth changes: instead, the rendering of the screen may need to change from hardware accelerated rendering to software.

A key enabler of this extension is the advent of software frame buffer rendering code (called “fb”, to distinguish it from the “mfb” and “cfb” implementations used in older X server implementations) that can support all depths simultaneously (while being dramatically more compact and as fast or faster than the old frame buffer code on current processors). Recent machines are fast enough for this to provide an adequate level of performance for all but the most performance critical applications.

The Render extension enables arbitrary conversions among pixel formats allowing, for example, the display of 24 bit RGB data on a 16 bit display. RandR needs this to convert all pixel formats to one supported by the display hardware.

RandR permits the list of visuals that are accelerated by the hardware to change on the fly. Toolkits and/or occasionally clients may want to follow this information and dynamically select which visual is being used. This extension can inform clients when such changes occur.

2 Migration and Replication

Migration takes an existing running application and transfers it’s display to another device. Replication takes that same application and duplicates it’s output (and input) on multiple devices.

2.1 Motivation

We believe the ability to migrate applications between X servers is a critical component for the vision of pervasive computing: applications should be able to migrate between displays routinely as users move and interact with handheld, desktop, appliance and projector screens in the global internet environment. Additionally, applications should be able to survive the loss of their server connection. At some future time they should reconnect to either the same or other device.

For example, you should be able to tell an application running on your handheld computer to use a nearby desktop display, keyboard and mouse, or a projector on the wall. This should not require stopping and starting the application. You should be able to go home, and decide to import applications you left running at work. There are obviously security, authentication and authorization problems left to work out, but these are generally independent of the base window system.

2.2 Difficulties Migrating X Applications

Migration and replication have traditionally been difficult in X due a number of interrelated factors:

- pseudocolor displays - There was no guarantee that the color you needed or even an approximation of it would be available on the destination server.
- pixmap depths - servers have typically only supported a few of the permitted possible pixmap depths; the software frame buffer code typically only supported 1, 8 and 32 bit displays, and would not be present if the server did not have the capability to be used at that depth.
- frame buffer depth - lack of any capability to emulate non-native frame buffer formats.

These taken together meant that applications and toolkits had to be carefully written to survive migration or

replication, and that applications on most common toolkits could not migrate at all. Retrofitting the toolkits was very difficult afterwards due to these issues. Server resources (e.g. pixmaps) might need recomputation to alternate depths, and applications often depend on particular visual types be available throughout their execution. Pixel values do not easily map between servers for pseudocolor visual types, and are not even present with pixmaps, which do not have inherent color information.

While migration of applications between X servers has always been possible in X, it was so difficult that unless prior thought were taken both toolkits and applications it would not occur. In practice, it is extremely rare, since toolkit writers did not think it important (at least at the beginning of their projects).

As a result, in practice, migration and replication has at best happened rarely and has been fraught with problems. Only a few research toolkits like Trestle [MN91] have been built with these capabilities, and the retrofit into existing toolkits would have been so difficult as to be impossible. In our opinion, this problem has been one of the most painful limitations in X11 protocol's design. Only a limited number of applications have been migratable and replicable, for example, GNU/emacs [Sta00] using the obscure "make-frame-on-display" function.

Proxy server approaches [GWY94] came closest to a general solution, but have serious drawbacks. For best performance, applications should be communicating directly with the X server without a proxy. The proxy must reformat much of the X protocol traffic and even still, ensuring that the result remains compliant with the X protocol specification is quite difficult. Even with these difficulties, proxy servers remain useful.

Another problem is the large variation of display sizes, applications that fit easily on a desktop screen may not fit on a handheld or appliance screen. This problem has intensified over the last 10 years as X servers have been deployed on progressively smaller platforms. Proxy server based replication can't make applications adapt to this change, only toolkit based solutions can. Replicated applications are the basis of important real time collaborative applications, and this ability should span handhelds, desktops and larger displays.

2.3 Promoting Toolkit Level Migration and Replication

The best solution is for applications and toolkits to support migration and replication themselves. Making it easy for existing toolkit implementers to implement migration and replication is key to migration and replication becoming ubiquitous. The solutions we have outlined provide for more uniformity among the capabilities of different X servers. This increased uniformity should simplify the implementation of migration and replication in existing toolkits and applications that were not designed to cope with multiple disparate X servers.

It has taken more than five years longer for pseudocolor to phase out than we had anticipated. Instead of increasing in capability, display hardware architecture remained relatively constant while costs reduced dramatically. Fortunately, most desktop environments are finally regularly running in truecolor mode. Very few applications now require pseudocolor visual types to function, whereas pseudocolor was dominant 10 years ago. The dominant applications that people care about now work with fixed color maps. The lack of pseudocolor displays simplifies the translation among color representations.

Additionally, the Render extension moves applications toward a more abstract representation of color, pixel values become much less important as applications focus on color while the extension automatically translates between the various representations of the data.

Modern toolkits like GTK+ 2.0 and Qt [Dal01] now isolate applications entirely from visual representations from X, and should become much easier to adapt to enable server migration and replication. This would enable all new X applications to be used in a fundamentally more interesting fashion. Even applications entirely based on Xt based widgets or other toolkit should be so migratable with some work, as we have carefully avoided violating the invariants found in most applications we are aware of. We strongly advocate the toolkit work to complete this vision.

RandR itself only provides a small part of the solution (the ability to determine which visuals are accelerated after a change event). The deployment of the RandR and the Render extensions in concert with "fb", but more importantly the internal X server implementation required to support them results in X servers offering all pixmap depths. With the phase out of pseudocolor visual types, this should dramatically reduce the variability of server

configurations encountered by toolkits. Rather than few if any servers supporting all depths, all servers will eventually support all depths. So we expect to see a great increase in uniformity between X server implementations as these servers deploy, greatly easing the problems faced by toolkit implementers. See the section below on Server implementation for details.

As of the time of this writing, we have not demonstrated application migration using these facilities. We plan to do so soon.

3 Description

Clients can select for `RRScreenChange` events to be informed if certain properties of a screen have changed.

The root depth, visual and colormap cannot change when a screen is reconfigured to avoid confusing naive clients. Instead, the X server may re-render onto a different depth of framebuffer, and therefore the visuals that can be accelerated by hardware may change. Clients may therefore wish to be informed when these changes occur, and select other visuals for rendering.

Any server supporting this extension will not list the same visual id at more than one depth. This is to make a visual id uniquely identify a depth.

The core protocol allows the same visual ID to be reported for multiple screens, but as the sample X server implementation does not do so and as there is no change in functionality by making this restriction, the RandR extension gains quite a bit of simplicity by enforcing this restriction.

Suppose a non-RandR-aware client has a window on a non-default depth 24 visual, and is switched to a higher resolution, where depth 24 is not supported. How can this work?

In this and similar cases, the window will be rendered using software and updated asynchronously to the display using the Render extension capabilities for blitting between depths. Note that your performance may suffer, although experience with the shadow frame buffer implementation in the XFree86 server shows that the hit is not as bad as you might think; the real frame buffer is never being read, only written.

For simplicity, RandR uses the “acceleration” Boolean

value instead of the more complex integer for visual quality that the Double Buffer Extension [EW94] uses. Applications will search for “accelerated” visuals which meet their requirements with the assumption that non-accelerated visuals will be implemented using the method described above.

4 Implementation

During the implementation of RandR within XFree86, several distinct issues needed addressing:

- **Multiple Depths.** RandR requires that all possible depths be available all of the time.
- **Rotation.** As hardware doesn’t normally support this, software will have to rotate all rasterization.
- **Size and Depth Changing.** Changing the size requires recomputation of window clip lists, changing the depth requires reprogramming the hardware.

Each was managed without significant new code by taking advantage of recent advancements within the XFree86 distribution.

4.1 Supporting Additional Depths

Because of the pixel-value oriented rasterization model in the core X protocol, the X server is essentially required to render using precisely the same pixel format as is advertised to applications. This implies that when the hardware frame buffer format does not match the visual, the server must store the pixels off screen in their advertised format. This ensures that raster operations that directly manipulate pixel values operate correctly and that applications can continue to use `GetImage` and retrieve precisely the right values.

This is implemented using the XFree86 shadow frame buffer code. The shadow frame buffer code was originally designed to allow simple X server porting to frame buffers using a format not supported by the frame buffer code. It works by creating a virtual frame buffer in application memory. All rendering operations are directed to this virtual frame buffer and the areas affected by these operations are tracked by the shadow frame buffer code. The hardware frame buffer is periodically updated by

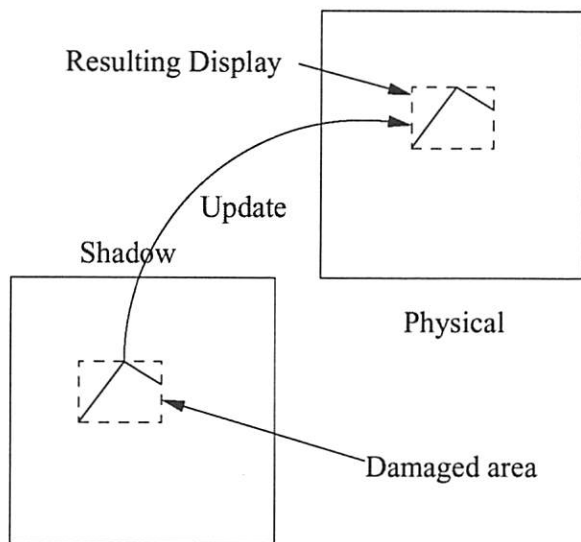


Figure 1: Shadow Frame Buffer Operation

copying data from the shadow frame buffer as seen in Figure 1.

The shadow frame buffer replaces the need for a complete rendering implementation with a simple copy operation from the virtual frame buffer to the hardware. While rendering within the shadow frame buffer cannot be accelerated with the video hardware, the fact that the frame buffer lives in regular memory and not across a PCI or AGP bus means that performance is generally acceptable. Because damaged regions are batched together during periodic updates, the bandwidth of the bus doesn't significantly impact overall performance.

RandR takes this shadow frame buffer implementation and creates a virtual frame buffer for each depth not supported by the hardware. As seen in Figure 2, copying the data from the virtual frame buffer to the hardware involves converting the format of the data to match the hardware, but as the original pixel data are preserved in the shadow frame buffer, the display remains correct even in the presence of complicated raster operations. Depths supported by the hardware remain implemented directly on the hardware with no intermediate shadow frame buffer. While this architecture could use the additional frame buffers to eliminate some window damage, the current implementation doesn't perform this optimization.

Using the shadow frame buffer in this way presumes that the X server can already render to the necessary formats. One recent addition to the XFree86 X server is "fb", a new implementation of simple frame buffer rendering

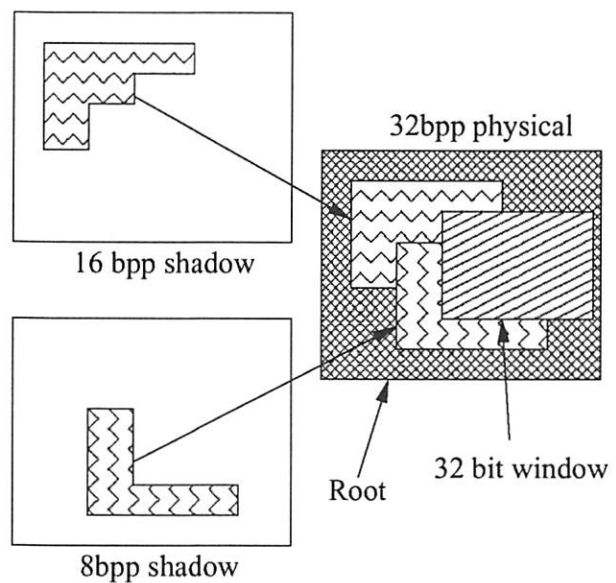


Figure 2: Supporting Multiple Depths

code. By taking advantage of changes in the relationship between CPU and memory performance, fb packs support for all X pixel sizes into a single implementation. This permits every X server to support all pixel sizes without any effort, something which had been quite difficult in the past. RandR takes advantage of this to provide simultaneous support for all of the displayable pixel sizes.

Displaying the contents of these shadow frame buffers involves converting the pixel values from one format to another while preserving the RGB values. The Render extension already provides this capability; all operations within that extension accept operands in arbitrary formats. RandR takes advantage of this functionality to perform the format conversion when updating the hardware frame buffer with the contents of the virtual frame buffers in.

4.2 Rotating the Screen

As hardware doesn't generally support changing the order that pixels are read out of memory when generating the screen image, rotating the screen involves changing the software to rotate the image within the frame buffer.

There are two possible implementations for this; the most obvious is to rotate all of the rendering requests directed at the screen. This turns out to be somewhat complicated as there are many operations that involve

interactions between the screen and off-screen data and each one must rotate the data.

While this may eventually be implemented, the current RandR implementation uses the shadow frame buffer code as described above to keep a copy of the screen in "normal" format; updates to the hardware frame buffer rotate the image to orient the view correctly. This implementation causes a slight loss of performance as no rendering is accelerated and also uses additional main memory for a copy of the frame buffer.

4.3 Changing the Screen Size and Depth

The initial implementation for RandR has been done using the Tiny-X architecture within the XFree86 X server. This alternate driver mechanism provides a minimal X server which eases development of new extensions involving interactions between the device drivers and the extension. RandR requires some significant interactions to switch screen sizes and depths. The intent is to migrate this architecture into the core XFree86 X server when the interfaces have stabilized.

Tiny-X already supports dynamically reconfiguring the size and depth of the video hardware; this capability is exported to the user by exposing multiple "screens" sharing the same physical video card. Activating a new screen involves disabling rendering to the old screen, reprogramming the video hardware and enabling rendering for windows on the new screen.

Extending this to support RandR is straightforward: disable rendering to the current screen, reprogram the hardware for the new configuration and re-enable rendering. When the depth changes, virtual frame buffers for the old hardware depth become the targets for rendering at those depths. The new depth receives direct access to the hardware while it's virtual frame buffer is disposed of.

4.4 X Library Implementation

We envision Xlib will always select for RRScreensChange events on all screens, and that it will change the screen structure automatically when such events arrive. (We will experimentally determine if this seems to confuse clients. By making the restriction that the default depth and visual cannot change when a screen is changed, we believe most clients will not need modification to support screen changes.)

We expect that most clients and toolkits will be oblivious to changes to the screen structure, as they generally use the values in the connections Display structure directly. By updating on the fly, we believe pop-up menus and other pop up windows will position themselves correctly in the face of screen configuration changes (the issue is ensuring that pop-ups are visible on the reconfigured screen).

Advanced toolkits may wish to use the facilities of the extension to determine which visuals are accelerated, and possibly change to use them.

5 The RandR Extension

The extension itself is short enough that the inclusion of most of it is worthwhile (we have elided the encoding). The version in this paper is Version .91, and is not (yet) a standard of any kind, and is certain to change before being finalized. We expect it is likely some additional change may be required as we gain implementation experience.

The specification methodology follows that of other X Window System protocol design specifications.

5.1 Types

The following types are used in the request and event definitions in subsequent sections:

ROTATION:	{ 0, 90, 180, 270 }. Degrees clockwise rotation from native frame buffer orientation
VISUALSET:	LISTofVISUALID
VISUALSETID:	CARD16 index into VISUALSET
SETofVISUALSET:	LISTofVISUALSETID
SETofVISUALSETID:	CARD16 index into SETofVISUALSET
SIZES:	[width-in-pixels: CARD16 height-in-pixels: CARD16 width-in-mm: CARD16 height-in-mm: CARD16 visual-group-id: SETofVISUALSETID]
SIZESSET:	LISTofSIZES

SIZESETID: CARD16 index into SIZE-SET

5.2 Requests

RRQueryVersion

clientMajorVersion: CARD16
clientMinorVersion: CARD16
→
serverMajorVersion: CARD16
serverMinorVersion: CARD16

The version numbers are an escape hatch in case future revisions of the protocol are necessary. The major version must increment for incompatible changes, and the minor version SHOULD increment for small upward compatible changes. Unrecognized requests or extra arguments to a request within a major version MUST be ignored. Barring changes, the major version will be 1, and the minor version will be 0.

A server may support multiple versions of the extension: the reported version is the one which will be used.

RRGetScreenInfo

drawable: Drawable
→
size-set: SIZESET
size-set-index: SIZESETID
visual-set: SETOFVISUALSET
visual-set-index: VISUALSETID
accelerated: LISTofVISUALSET
rotations-possible: LISTofROTATIONS
rotation: ROTATION
timestamp: TIMESTAMP
Errors: Drawable

The set of visuals advertised by the X server can never change. This is to prevent confusion of naive applications that may presume that the visual type and depth of the root is fixed for all time: instead, the X server will rerender the windows on a changed depth, though performance may be degraded unless the client afterwards selects a visual known to be hardware accelerated.

Accelerated is a list of the various possible configurations for hardware assisted visual support on the screen associated with the specified drawable. Hardware assisted visuals can be expected to be more efficient than

non-assisted visuals and may provide more timely feedback for graphics requests.

The visual-set-index indicates which visual set is currently being used to render the screen.

This request returns possible size configurations on the screen associated with the specified drawable.

Modern toolkits SHOULD use RHardwareSelectInput to be notified via a RRHardwareVisualChange event, so that they can change visual types and/or depths and continue to receive the benefits of hardware acceleration.

size-set-index, visual-set-index, rotation, indicate which sizeset entry is being used, which visualset entry is being used, the current screen rotation.

The rotations-possible values indicate which rotations are supported by this server for this screen.

The timestamp indicates when the size-set information last changed: requests to set the screen will fail unless the timestamp indicates that the information the client is using is up to date, to ensure clients can be well behaved in the face of race conditions.

Note that much of this information is provided to enable hot-swapping of display cards, which can already occur on handheld computers using PCMCIA and may occur in the future on other busses. Our experience over X's history is that items we thought were static have often become dynamic as technology changes, so we are designing with the presumption that almost anything about the X server could change.

RRSetScreenConfig

draw: DRAWABLE
size-set-index: SIZESETID
visual-set-index: VISUALSETID
rotation: ROTATION
timestamp: TIMESTAMP
→
new-timestamp: TIMESTAMP
Errors: Value, Match, Drawable

Sets the screen to the specified element of the screen set list returned from RRGetScreenInfo, and the specified hardware visual set element.

Sets the hardware visual set to the specified element of the visualsets list from RRQueryHardwareVisuals.

The screen may be rotated by the specified rotation.

If the timestamp of this request is less than the timestamp reported in the latest `RRScreenChange` event send, the request is ignored. (This could occur if the screen changed since you last made a `RRGetScreenInfo` request. Rather than allowing an incorrect call to be executed based on stale data, the server will ignore the request.) The new-time-stamp is returned: if it is not equal to the timestamp you provided, you know the request failed and your screen information is stale and should be re-read.

RRScreenChangeSelectInput

window: WINDOW

enable: BOOL

Errors: Value, Window

Requests that `RRScreenChange` events of screen changes of the screen associated with the drawable be delivered to the specified window. (whew!)

Clients may then choose to create new resources or use other visuals that have hardware acceleration available that take advantage of the new screen configuration.

5.3 Events

To reconfigure the root window, use the `RRSetScreen` request defined above.

RRScreenChangeNotify

root: WINDOW

size-set-index: SIZESETID

visual-set-index: VISUALSETID

rotation: ROTATION

hardware: BOOL

config-timestamp: TIMESTAMP

timestamp: TIMESTAMP

This event is delivered to clients selecting for notification with `RRScreenChangeSelectInput` requests.

This event is sent whenever the screen is changed, or if a new screen configuration becomes available that was not available in the past. The client **MUST** call `RRGetScreenInfo` to update its view of possible screen configurations (and get an up to date timestamp required by the `RRSetScreen` request).

This event is delivered to clients selecting for notification with `RRScreenChangeSelectInput` to the window. *visual-set-index* indicates the current visual set and hardware indicates if the specified window's visual is a member of that set. The hardware boolean indicates if the selected window's visual is still accelerated by hardware.

6 History and Status

This extension has been designed and significant external review input incorporated. A prototype implementation is functioning in the TinyX X implementation.[CP01]

7 Future Work

The XAA [VF00] implementation of the full XFree86 will need extension to fully support this protocol extension, prototyped in the TinyX framework.

Toolkits will need updating to support RandR to take advantage of accelerated visual type information to ensure highest possible performance.

Window managers will need to support this extension to layout the screen in some fashion when the size changes to ensure applications are appropriately visible.

There are very entertaining user interface possibilities for moving applications between screens, particularly once systems become aware of resources available in their nearby environment. A free, off the wall (or maybe on the wall), idea might be given a handheld computer with accelerometers such as Itsy[HWV⁺01] you might almost literally throw windows from one screen to another. Other hacks are left to the bizarre nature of your own imagination.

Acknowledgments

The authors would like to thank their respective employers, Compaq and SuSE, for support of open source software development and the XFree86 project for its continuing advancement of the X Window System. Additional thanks go to Carl Worth and Alexander Guy for

their help in editing the manuscript.

References

- [CP01] Juliusz Chroboczek and Keith Packard. *Xkdrive - Tiny X server*. The XFree86 Project, Inc., 2001. XFree86 Release 4.0.3.
- [Dal01] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, May 2001.
- [EW94] Ian Elliott and David P. Wiggins. Double Buffer Extension Protocol. X consortium standard, X Consortium, Inc., 1994.
- [GWY94] Daniel Garfinkel, Bruce C. Welte, and Thomas W. Yip. HP SharedX: A Tool for Real-Time Collaboration. *HP Journal*, April 1994.
- [HWV⁺01] William R. Hamburg, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the Bounds of Mobile Computing. *IEEE Computer*, 34(4):28–35, April 2001.
- [MN91] Mark Manasse and Greg Nelson. Trestle Reference Manual. Research Report 68, Digital Equipment Corporation Systems Research Center, December 1991.
- [SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.
- [Sta00] Richard M. Stallman. *GNU Emacs Manual*. Free Software Foundation, fourteenth edition, June 2000.
- [VF00] Mark Vojkovich and Marc Aurele La France. XAA.HOWTO. Technical report, The XFree86 Project, Inc., 2000.

MEF: Malicious Email Filter

A UNIX Mail Filter that Detects Malicious Windows Executables

Matthew G. Schultz and Eleazar Eskin

*Department of Computer Science
Columbia University*

{mgs,eeskin}@cs.columbia.edu

Erez Zadok

*Department of Computer Science
State University of New York at Stony Brook*

ezk@cs.sunysb.edu

Manasi Bhattacharyya, and Salvatore J. Stolfo

*Department of Computer Science
Columbia University*

{mb551,sal}@cs.columbia.edu

Abstract

We present *Malicious Email Filter*, MEF, a freely distributed malicious binary filter incorporated into Procmail that can detect malicious Windows attachments by integrating with a UNIX mail server. The system has three capabilities: detection of known and unknown malicious attachments, tracking the propagation of malicious attachments and efficient model update algorithms.

The system filters multiple malicious attachments in an email by using detection models obtained from data mining over known malicious attachments. It leverages preliminary research in data mining applied to malicious executables which allows the detection of previously unseen, malicious attachments. In addition, the system provides a method for monitoring and measurement of the spread of malicious attachments. Finally, the system also allows for the efficient propagation of detection models from a central server. These updated models can be downloaded by a system administrator and easily incorporated into the current model. The system will be released under GPL in June 2001.

1 Introduction

A serious security risk today is the propagation of malicious executables through email attachments. A malicious executable is defined to be a program that performs a malicious act, such as compromising a system's security, damaging a system or obtaining sensitive information without the user's permission. Recently there have been some high profile incidents with malicious email attachments such as the ILOVEYOU virus and its clones. These malicious attachments caused significant damage in a short time. The *Malicious Email Filter* (MEF) project provides a tool for the protection of systems against malicious email attachments.

An email filter that operates within a mail to detect malicious Windows binaries has many advantages. Op-

erating from a mail server, the email filter could automatically filter the email each host receives. The mail server could either wrap the malicious email with a warning addressed to the user, or it could block the email depending upon the server's settings. All of this could be done without the server's users having to scan attachments themselves or having to download updates for their virus scanners. This way the system administrator can be responsible for updating the filter instead of relying on end users. We present such a system on a UNIX implementation of sendmail using procmail.

The standard approach to protecting against malicious emails is to use a virus scanner. Commercial virus scanners can effectively detect known malicious executables, but unfortunately they can not detect unknown malicious executables reliably. The reason for this is that most of these virus scanners are *signature based*. For each known malicious binary, the scanner contains a byte sequence that identifies the malicious binary. However, an unknown malicious binary, one without a pre-existing signature, will most likely go undetected.

We built upon preliminary research at Columbia University on data-mining methods to detect malicious binaries [2]. The idea is that by using data-mining, knowledge of known malicious executables can be generalized to detect unknown malicious executables. Data mining methods are ideal for this purpose because they detect patterns in large amounts of data, such as byte code, and use these patterns to detect future instances in similar data along with detecting known instances. Our framework used *classifiers* to detect malicious executables. A classifier is a rule set, or detection model, generated by the data mining algorithm that was trained over a given set of training data.

The goal of this paper is to describe a data mining based filter which integrates with Procmail's pre-existent security filter [3] to detect malicious executables. The MEF system is an application of more theoretical research into this problem [10]. The data mining-based de-

tection system within MEF is a preliminary system that will become more accurate and efficient as our research progresses, and new data sets are analyzed. It uses a scoring system based on a data mining classifier to determine whether or not an attachment may be malicious. If an attachment's score is above a certain threshold it is considered malicious.

This work expanded upon Procmail's pre-existent filter which already defangs active-content HTML tags to protect users who read their mail from a web browser or HTML-enabled mail client. Also, if the attachment is labeled as malicious, the system "mangles" the attachment name to prevent the mail client from automatically executing the attachment. It also has built in security filters such as long filenames in attachments, and long MIME headers, which may crash or allow exploits of some clients. This filter lacks the ability to automatically update its list of known malicious executables leaving the system vulnerable to attacks by new and unknown viruses. Furthermore, its evaluation of an attachment is based solely on the name of the executable and not the contents of the attachment itself. We replaced this signature based detection algorithm with our data mining classifier that added the ability to detect both the set of known malicious binaries and a set of previously unseen, but similar malicious binaries. Although the MEF implementation was designed for the data mining-based detection system, any method to evaluate binaries including a standard signature based scanner can be used.

Since the methods and classifier models we describe are probabilistic, we provide a means of determining whether a binary was *borderline*. A borderline binary is a program that has similar probabilities for both classes (i.e. could be either a malicious executable or a benign program). As a parameter of the filter, the system administrator may specify what a borderline case is. Guidelines on how to set this parameter are described in detail later. If it is a borderline case then along with the option to wrap it as a malicious program there is an option in the network filter to send a copy of the malicious executable to a central repository such as CERT. There, it can be examined by human experts. After analysis by virus experts, the model can be updated to be more accurate by including these borderline cases.

The detection model generation works as follows. The binaries are first statically analyzed to extract byte-sequences, and then the classifiers are generated by analyzing a subset of the data. Then the classifier (or detection model) is tested on a set of previously unseen data. We implemented a traditional, signature-based algorithm to compare its performance with the data mining algorithms. Using standard statistical cross-validation techniques, the data mining-based framework for malicious binary detection had a detection rate of 97.76%, over

double the detection rate of a signature-based scanner.

The organization of the paper is as follows. In Section 2, we present the system features and their integration with Procmail. In Section 3, we detail the methods that are employed to track the propagation of malicious attachments. Section 4 describes how the detection algorithms work, and their results. Section 5 discusses the system's performance, and Sections 6 and 7 conclude the paper and discuss future work.

2 Incorporation into Procmail

MEF filters malicious attachments by replacing the signature based virus filter found in Procmail with a data mining generated detection model. Procmail is a program that processes email messages looking for particular information in the headers or body of each message, and takes actions based on what it finds [11]. Currently the mail server supported is sendmail. MEF uses a procmail script to extract attachments from emails and save them temporarily based on their name. The script then runs the filter on each attachment.

The filter first decodes each binary and then examines the binary using a data mining classifier. It evaluates the attachment by comparing it to all the byte strings found with it to the byte-sequences contained in the detection model. The system calculates the probability of the binary being malicious, and if it is greater than its likelihood of being benign then the executable is labeled malicious. Otherwise, the binary is labeled benign. This is reported as a score back to Procmail, and then is used to either send the mail along untouched, or the entry is logged as the attack and email is wrapped with a warning. The log is a collection of information about the attachment. Exactly what this information is depends upon the configuration of the system.

2.1 Borderline Cases

Borderline binaries are binaries that have similar probabilities of being benign and malicious (e.g. 50% chance it is malicious, and 50% chance it is benign). The binaries are important to keep track of because they are likely to be mislabeled, so they should be included in the training set. To facilitate this, the system archives the borderline cases, and at periodic intervals the collection of borderline binaries is sent back to a central server by the system administrator.

Once at the central repository, these binaries can then be analyzed by experts to determine whether they are malicious or not, and subsequently included in the future versions of the detection models. Any binary that is determined to be a borderline case will be forwarded to the

repository and wrapped with a warning as though it were a malicious attachment.

A simple metric to detect borderline cases and redirect them to an evaluation party is to define a borderline case to be a case where the difference between the probability it is malicious and the probability it is benign is above a threshold. This threshold is set based on the policies of the host.

For example in a secure setting, the threshold could be set at 20%. In this case all binaries that have a 60/40 split are labeled as borderline. In other words, binaries with a 60% chance (according to the model) of being malicious and 40% chance of being benign would be labeled borderline, and vice versa. This setting can be determined by the system administrator or left on the default setting of 51.25/48.75, a threshold of 2.5%.

Receiving borderline cases and updating the detection model is an important aspect of the data mining approach. The larger the data set that is used to generate models then the more accurate the detection models will be. This is because borderline cases are executables that could potentially lower the detection and accuracy rates by being misclassified, so they should be trained over.

2.2 Update Algorithms

This system will require updates periodically, and in the following section we detail the update algorithm. After a number of borderline cases have been received, it is necessary to generate a new detection model, and subsequently distribute updated models.

A new model is first generated by running the data mining algorithm on the new data set that contained the borderline cases along with their correct classification, and the previous data set. This model will then be distributed.

Updating the models is accomplished by distributing portions of the models that changed, and not the entire model. This is important because the detection models are large. In order to avoid constantly sending a large model to the filters, the administrator has the option of receiving this smaller file. Using the update algorithm, the older model can then be updated. The full model will also be available to provide additional options for the system administrator.

Efficient update of the model is possible because the underlying representation of the models is probabilistic. As is explained later, the model is a count of the number of times that each byte string appears in a malicious program versus the number of times that it appears in a benign program. An update model can then be easily summed with the older model to create a new model.

In future versions of MEF, the model will be made available for the system administrator on a public ftp site.

If a system administrator subscribes to the mailing list then when a new model is made available, the system administrator will receive an email. The email will detail where the model is located, what version it is, and include a form of authentication. At the ftp site the model will be available to download as either an upgrade from a previous version, or as a full model. An archive of old models will also be kept on the ftp site.

There are also a host of options for automatically receiving the updates. One way to distribute the email is just to attach the update to the notification email. Then the administrator could update the model later without having to ftp it. In the future, a program included in the email filter could automatically poll the central server to see if a new model is available and then download it and update the current model. These last methods have not yet been implemented.

3 Monitoring the Propagation of Email Attachments

Tracking the propagation of email attachments is beneficial in identifying the origin of malicious executables, and in estimating a malicious attachment's prevalence.

The monitoring works by having each host that is using the system log all malicious attachments, and the borderline attachments that are sent to and from the host. This logging may or may not contain information about the sender or receiver depending on the privacy policy of the host.

In order to log the attachments, we needed a way to obtain a unique identifier for each attachment. We did this by using the MD5 algorithm [9] to compute a unique number for each binary attachment. The input to MD5 was the hexadecimal representation of the binary. These identifiers were then kept in a log along with other information such as whether the attachment was malicious, or benign and with what certainty the system made those predictions.

The logs of malicious attachments are then sent back to the central server according to the policy of each host. Some hosts may wish to never send these logs, and can turn the feature off, while other hosts could configure the system to only send logs of borderline cases, etc.

After receiving the logs, the system measures the propagation of the malicious binaries across hosts. From these logs it can be estimated how many copies of each malicious binary were circulating the Internet, and these reports will be forwarded back to the community, and used for further research.

The current method for detailing the propagation of malicious executables is for an administrator to report an attack to an agency such as *WildList* [8]. The wild list

is a list of the propagation of viruses in the wild and a list of the most prevalent viruses. This is not done automatically, but instead is based upon a report issued by an attacked host. Our method would reliably, and automatically detail a malicious executable's spread over the Internet.

4 Methodology for Building Data Mining Detection Models

We gathered a large set of programs from public sources and defined a learning problem with two classes: *malicious* and *benign* executables. Each example in the data set is a Windows or MS-DOS format executable, although the framework we present is applicable to other formats. To standardize our data-set, we used McAfee's [6] virus scanner and labeled our programs as either malicious or benign executables.

We split the dataset into two subsets: the *training set* and the *test set*. The data mining algorithms used the training set while generating the rule sets, and after training we used a test set to test the accuracy of the classifiers on a set of unseen examples.

4.1 Data Set

The data set consisted of a total of 4,301 programs split into 3,301 malicious binaries and 1,000 benign programs. The malicious binaries consisted of viruses, Trojans, and cracker/network tools. There were no duplicate programs in the data set and every example in the set is labeled either malicious or benign by the commercial virus scanner. All labels are assumed to be correct.

All programs were gathered either from FTP sites, or personal computers in the Data Mining Lab at Columbia University. To obtain the dataset please contact us through our website <http://www.cs.columbia.edu/ids/mef/>.

4.2 Detection Algorithms

We statically extracted byte sequence *features* from each binary example for the learning algorithms. Features in a data mining framework are properties extracted from each example in the data set, such as byte sequences, that a classifier uses to generate detection models. These features are then used by the algorithms to generate detection models. We used *hexdump* [7], an open source tool that transforms binary files into hexadecimal files. After we generated the hexdumps we produced features in the form displayed in Figure 1 where each line represents a short sequence of machine code instructions.

```
646e 776f 2e73 0a0d 0024 0000 0000 0000
454e 3c05 026c 0009 0000 0000 0302 0004
0400 2800 3924 0001 0000 0004 0004 0006
000c 0040 0060 021e 0238 0244 02f5 0000
0001 0004 0000 0802 0032 1304 0000 030a
```

Figure 1: Example Set of Byte Sequence Features

4.3 Data Mining Approach

The classifier we incorporated into Procmail was a Naive Bayes classifier [1]. A naive Bayes classifier computes the likelihood that a program is malicious given the features that are contained in the program. We assumed that there were similar byte sequences in malicious executables that differentiated them from benign programs, and the class of benign programs had similar sequences that differentiated them from the malicious executables.

The model output by the naive Bayes algorithm labels examples based on the byte strings that they contain. For instance, if a program contained a significant number of malicious byte sequences and a few or no benign sequences, then it labels that binary as malicious. Likewise, a binary that was composed of many benign features and a smaller number of malicious features is labeled benign by the system.

The naive Bayes algorithm computed the probability that a given feature was malicious, and the probability that a feature was benign by computing statistics on the set of training data. Then to predict whether a binary, or collection of hex strings was malicious or benign, those probabilities were computed for each hex string in the binary, and then the Naive Bayes independence assumption was used. The independence assumption was applied in order to efficiently compute the probability that a binary was malicious and the probability that the binary was benign.

One draw back of the naive Bayes method was that it requires more than 1 GB of RAM to generate its detection model. To make the algorithm more efficient we divided the problem into smaller pieces that would fit in memory and generated a classifier for each of the subproblems. The subproblem was to classify based on every 16 subsets of the data organized according to the first letter of the hex string.

For this we trained sixteen Naive Bayes classifiers so that all hex strings were trained on. For example, one classifier trained on all hex strings starting with an "A", and another on all hex strings starting with a "0". This was done 16 times and then a voting algorithm was then used to combine their outputs.

A more thorough description along with an example, can be found in a companion paper [10].

4.4 Signature-Based Approach

To compare our results with traditional methods we implemented a signature based method. First, we calculated the byte-sequences that were only found in the malicious executable class. These byte-sequences were then concatenated together to make a unique signature for each malicious executable example. Thus each malicious executable signature contained only byte-sequences found in the malicious executable class. To make the signature unique, the byte-sequences found in each example were concatenated together to form one signature. This was done because a byte-sequence that was only found in one class during training could possibly be found in the other class during testing [4], and lead to false positives when deployed.

The virus scanner that we used to label the data set contained signatures for every malicious example in our data set, so it was necessary to implement a similar signature-based method. This was done to compare the two algorithms' accuracy in detecting new malicious executables. In our tests the signature-based algorithm was only allowed to generate signatures for the same set of training data that the data mining method used. This allowed the two methods to be fairly compared. The comparison was made by testing the two methods on a set of binaries not contained in the training set.

4.5 Preliminary Results

To quantify the performance of our method we computed statistics on the performance of the data mining-based method versus the signature-based method. We are interested in four quantities in the experiments. They are the counts for *true positives*, *true negatives*, *false positives*, and *false negatives*. A true positive, TP, is an malicious example that is correctly tagged as malicious, and a true negative, TN, is a benign example that is correctly classified as benign. A false positive, FP, is a benign program that has been mislabeled by an algorithm as malicious, while a false negative, FN, is a malicious executable that has been mis-classified as a benign program.

The *overall accuracy* of the algorithm is calculated as the number of programs the system classified correctly divided by the total number of binaries tested. The *detection rate* is the number of malicious binaries correctly classified divided by the total number of malicious programs tested.

We estimated our results for detecting new executables by using 5-fold cross validation [5]. Cross validation is the standard method to estimate the performance of predictions over unseen data. For each set of binary profiles we partitioned the data into five equal size partitions. We used four of the partitions for training a model and then

evaluated that model on the remaining partition. Then we repeated the process five times leaving out a different partition for testing each time. This gave us a reliable measure of our method's accuracy on unseen data. We averaged the results of these five tests to obtain a measure of how the algorithm performs in detecting new malicious executables.

4.6 Performance on New Executables

Table 1 displays the results. The data mining algorithm had the highest detection rate, 97.76% compared with the signature based method's detection rate of 33.96%. Along with the higher detection rate the data mining method had a higher overall accuracy, 96.88% vs. 49.31%. The false positive rate at 6.01% though was higher than the signature based method, 0%.

Figure 2 displays the plot of the detection rate vs. false positive rate using *Receiver Operation Characteristic* curves [13]. Receiver Operating Characteristic (ROC) curves are a way of visualizing the trade-offs between detection and false positive rates. In this instance, the ROC curve show how the data mining method can be configured for different environments. For a false positive rate less than or equal to 1% the detection rate would be greater than 70%, and for a false positive rate greater than 8% the detection rate would be greater than 99%.

Profile Type	Detection Rate	False Positive Rate	Overall Accuracy
Signature Meth.	33.96%	0%	49.31%
Data Mining Meth.	97.76%	6.01%	96.88%

Table 1: The results of testing the algorithms over new examples. Note the Data Mining Method had a higher detection rate and accuracy while the Signature based method had the lowest false positive rate.

4.7 Performance on Known Executables

We also evaluated the performance of the models in detecting known executables. For this task, the algorithms generated detection models for the entire set of data. Their performance was then evaluated by testing the models on the same training set.

As shown in Table 2, both methods detected over 99% of known executables. The data mining algorithm detected 99.87% of the malicious examples and misclassified 2% of the benign binaries as malicious. However, we have the signatures for the binaries that the data mining algorithm misclassified, and the algorithm can include those signatures in the detection model without lowering accuracy of the algorithm in detecting unknown binaries. After the signatures for the executables that were

Profile Type	Detection Rate	False Positive Rate	Overall Accuracy
Signature Meth.	100%	0%	100%
Data Mining Meth.	99.87%	2%	99.44%

Table 2: Results of the algorithms after testing on known programs. In this task both algorithms detected over 99% of known malicious programs. We explain later the data mining algorithm can be boosted to have 100% accuracy by using some signatures.

misclassified during training had been generated and included in the detection model, the data mining model had a 100% accuracy rate when tested on known executables.

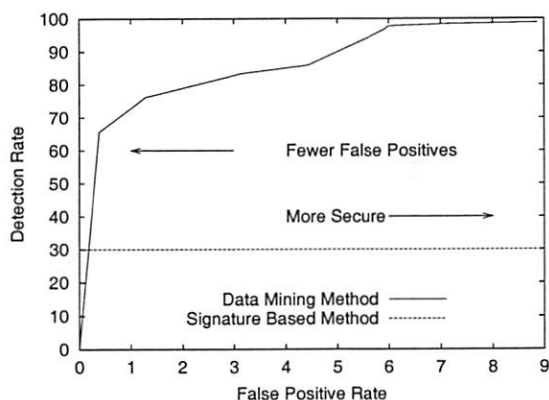


Figure 2: ROC: This figure shows how the data mining method can be configured to have different detection and false positive rates by adjusting the threshold parameter. More secure settings should choose a point on the data mining line towards the right, and applications needing fewer false alarms should choose a point towards the left.

5 System Performance

The system requires different time and space complexities for model generation and deployment.

5.1 Training

In order for the data mining algorithm to quickly generate the models, it requires all calculations to be done in memory. The algorithm consumed space in excess of a gigabyte of RAM. By splitting the data into smaller pieces, the algorithm was done in memory with no loss in accuracy.

The training of a classifier took 2 hours 59 minutes and 49 seconds running on a Pentium III 600 Linux machine with 1GB of RAM. The classifier took on average

2 minutes and 28 seconds for each of the 4,301 binaries in the data set.

5.2 During Deployment

Current work is being done to make the system accurate on a system with smaller memory. At this point in development, only systems that have a 1GB or more of memory can use our models. The amount of system resources taken for using a model are equivalent to the requirements for training a model. So on a Pentium III 600 Linux box with 1GB of RAM it would take on average 2 minutes 28 seconds per attachment.

The ongoing work we are doing is to make the model small enough to be loaded into a computer with 128MB of RAM without losing more than 5% in accuracy. If this is accomplished then the resources required in CPU time and memory will be notably reduced.

There are other options in making the system perform its analysis faster such as sharing the load over several computers. These options are not currently being explored, but they are open problems that the community should examine. We are primarily concerned with improving the space complexities of the algorithm without sacrificing a significant amount accuracy.

6 Conclusions

The first contribution that we presented in this paper was a freely distributed filter for Procmail that detected known malicious Windows executables and previously unknown malicious Windows binaries from UNIX. The detection rate of new executables was over twice that of the traditional signature based methods, 97.76% compared with 33.96%.

One problem with traditional, signature-based methods is that in order to detect a new malicious executable, the program needs to be examined and a signature extracted from it and included in the anti-virus database. The difficulty with this method is that during the time required for a malicious program to be identified, analyzed and signatures to be distributed, systems are at risk from that program. Our methods may provide a defense during that time. With a low false positive rate the inconvenience to the end user would be minimal while providing ample defense during the time before an update of models is available.

Virus scanners are updated about every month, and 240–300 new malicious executables are created in that time (8–10 a day [12]). Our method may catch roughly 216–270 of those new malicious executables without the need for an update whereas traditional methods would catch only 87–109. Our method tested on a particular

data set more than doubles the detection rate of signature based methods.

Secondly, we presented a system that improves its accuracy by regenerating models after receiving borderline cases. This feature is of interest because as more servers and clients use this system the system will receive additional borderline cases. Training on these borderline cases will increase the accuracy of the filter.

Finally, the system has the optional ability to monitor the propagation of malicious attachments. Depending upon the user specified setting, email tracking can be turned on or off. If tracking is turned on then statistics can be generated detailing how a malicious binary attacked a system and propagated. If tracking is turned off then the system loses no accuracy in detecting malicious attachments.

The system that we presented detected malicious Windows binaries from UNIX, and detected new examples of similar malicious binaries because of the data mining algorithms. It tracked the propagation of email attachments, and with the inclusion of borderline cases it will become more accurate with time. Also with a larger, more realistic data set work can be done to show the algorithm is practical.

7 Future Work

One of the most important areas of future work for this application is the development of more efficient algorithms. The current probabilistic method requires a machine with a significant amount of memory to generate, and employ the classifiers. This memory requirement makes the computation of the models expensive.

To make the algorithms use less space will require theoretical bounds on how to prune features from the data without losing accuracy. The details of how the pruning may work is beyond the scope of this paper.

After developing more efficient algorithms, the next most important work to be done is generating a more complete data set. The current, malicious data set, 3,301 examples, is smaller than the known number of malicious programs, 50,000+ examples. Work needs to be done with industry or security sources to develop a standard data set consisting of infected programs, macro and visual basic viruses, and many different sets of benign data.¹

On more obvious future work would be to incorporate the system into Windows and Macintosh mail servers and clients. This would require work with the individual vendors because their systems are not open-sourced. As a

¹We are currently working on establishing such a data set with Cigital, <http://www.cigital.com>, but more resources are needed. If you would like to work with us please contact us at our website <http://www.cs.columbia.edu/ids/mef>.

result of our system being freely available, these vendors could work with us to incorporate it or they could do it themselves.

Another potential future work of this filter is to make it into a stand alone virus scanner. Once the system has been fully completed, and thoroughly tested in the real world it would be possible to port the algorithms to different operating systems, such as Windows, or Macintosh.

This scanner could be run in a similar manner to traditional virus scanners. A user could run the system at bootup, or when required and analyze all the files on a personal computer. This requires though that the system be efficient enough to run on older computers with slower processors, and less memory.

8 Software Availability

The software described in this paper can be downloaded from <http://www.cs.columbia.edu/ids/mef> in June 2001.

References

- [1] D.Michie, D.J.Spiegelhalter, and C.C.Taylor. *Machine learning of rules and trees*. Ellis Horwood, 1994.
- [2] MEF Group. Malicious Email Filter Group. Published as *Online publication*, 2001. <http://www.cs.columbia.edu/ids/mef>.
- [3] John Hardin. Enhancing E-Mail Security With Procmail. *Online publication*, 2001. <http://www.impsec.org/email-tools/procmail-security.html>.
- [4] Jeffrey O. Kephart and William C. Arnold. Automatic Extraction of Computer Virus Signatures. *4th Virus Bulletin International Conference*, pages 178-184, 1994.
- [5] R Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *IJCAI*, 1995.
- [6] MacAfee. Homepage - MacAfee.com. Published as *Online publication*, 2001. <http://www.mcafee.com>.
- [7] Peter Miller. Hexdump. Published as *Online publication*, 2001. <http://www.pcug.org.au/millerp/hexdump.html>.
- [8] Wildlist Organization. Virus description of viruses in the wild. Published as *Online Publication*, 2001. <http://www.f-secure.com/virus-info/wild.html>.

- [9] Ronald L. Rivest. The MD5 Message Digest Algorithm. Published as *Internet RFC 1321*, April 1992.
- [10] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data Mining Methods for Detection of New Malicious Executables. *To appear in IEEE Symposium on Security and Privacy*, May 2001.
- [11] Stephen R. van den Berg and Philip Guenther. Procmail. *Online publication*, 2001. <http://www.procmail.org>.
- [12] Steve R. White, Morton Swimmer, Edward J. Pring, William C. Arnold, David M. Chess, and John F. Morar. Anatomy of a Commercial-Grade Immune System. *IBM Research White Paper*, 1999. <http://www.av.ibm.com/ScientificPapers/White/Anatomy/anatomy.html>.
- [13] KH Zou, WJ Hall, and D Shapiro. Smooth non-parametric ROC curves for continuous diagnostic tests. *Statistics in Medicine*, 1997.

Cost Effective Security for Small Businesses: A Guide to Open Source Solutions

Sean R. Brown

Applied Geographics, Inc.
Boston, MA

srbrown@appgeo.com

Abstract

As high bandwidth Internet access becomes more readily available at lower cost, many small companies are leveraging the Internet to expand their market share and grow their business. Companies choosing to connect their internal LANs to the Internet often sacrifice the security of local network resources for the sake of expediency and cost. With the availability of inexpensive hardware and the proliferation of open source software projects, highly reliable network security solutions are no longer just for large corporations.

The object of this paper is to provide a general overview of how a small or medium sized business can implement advanced and highly reliable network security solutions using freely distributable and open source software.

1. Introduction

In June of 1999 a destructive worm was released into the wild affecting users of the Microsoft Windows based operating system and the Outlook E-mail client. The worm, known as ExploreZip [1], was one in a series of viruses and worms to exploit known vulnerabilities in Windows network filesharing and Outlook attachment processing [2]. The ExploreZip worm spread rampantly throughout the US leaving many company LANs and personal home computers unusable or severely incapacitated. It destroyed or zeroed out random Microsoft Office documents on user and fileserver hard drives before spreading itself to other susceptible hosts. By some estimates [3] the ExploreZip worm caused up to \$7.6 billion damage before slowly fading away. The author of this worm was never caught.

In February of 2000, Yahoo, Amazon.com, Ebay, Etrade, and several other high profile ecommerce sites were targeted by a series of distributed denial of service (DDOS) attacks severely affecting site performance and leaving some sites unreachable [4].

The computer systems used in the attacks were largely owned by individuals, small businesses, universities and large companies, all with dedicated high bandwidth internet connections. Compromised through known vulnerabilities and published exploits the resulting network of infected host systems provided the platform from which the DDOS attacks were launched. In most cases, system owners had no idea that their computers were being used in the attacks.

Recent vulnerabilities in ISC's Bind, Washington University's wu-ftpd, lprng, and the ubiquitous rpc.statd resulted in the spread of three new Linux worms since January 2001 [5][6][7]. These worms perform a root compromise on the vulnerable system allowing the attacker complete control thus providing the launchpad for future attacks. Like the distributed denial of service attacks which struck in February 2000, these compromised systems are capable of forming networks creating the potential for additional DDOS attacks.

The above examples make it abundantly clear that businesses need to take responsibility for protecting themselves and their bottom line from malicious intruders. The rush to get connected was pursued without understanding the potential dangers of being online. Now that they are online, many businesses get buried under the added cost of cleaning up after the latest virus or recovering from a denial of service attack. While the costs associated with these attacks can often be mitigated over the short term, their impact can persist for months and even years as corrupt data, destroyed files and, in some cases, lost business.

While there are many ways to protect networks from external and internal attack, the simple fact is, many small businesses do not have not the financial resources required to implement costly hardware and software security solutions. However, there are numerous freely available security solutions that can, at a minimum, eliminate the upfront software costs of protecting small business networks.

This does not suggest that installing some free software will make a company's network security problems disappear. First, network security is a process which requires a change in mindset about how a business interfaces with the rest of the world. Just as you would not leave the front door to the office open and unlocked when no one is at work, you should likewise, not leave the database server containing all of your companies financial and accounting data connected directly to the internet. Second, any security based software you install, whether it be a proprietary, closed source firewall or a free, open source intrusion detection system, will require configuration, installation and monitoring. Like it or not, the price of network security becomes the cost of doing business online, no matter what the marketing engines say.

The focus of this paper is threefold; first, to look at the various open source or free software available for protecting small networks, second, to describe a case study in open source network security, and third, to summarize the effectiveness of open source firewalling from the perspective of the case study.

2. Tools

While there is an abundance of open source and free software available for securing networks, I will focus on those which I have used in production environments. I will arbitrarily categorize the tools by function in the following way:

- Firewall
- System Integrity/Host Intrusion Detection
- Central Logging
- Encryption Software and Protocols
- Network Intrusion Detection

Obviously, the technological environment in which we live is constantly changing. Software developers are continually updating existing tools and creating new ones to address security threats. While there are a number of tools discussed in this paper, it is by no means, an exhaustive list, nor is that the intent. Rather, this should be considered a primer on some common solutions and a reference for further investigation.

2.1 Firewall

In any networked environment, threats to system integrity come from sources both external and internal to that network. At its most fundamental level a firewall is intended to mitigate external threats by monitoring all traffic entering or leaving a network

and allowing or denying that traffic based on packet content. As such a firewall is the first line of defense against external threats.

There are two basic types of firewalls: packet filters [8] and application layer gateways or proxies [9]. A packet filter inspects each packet at the network and transport layers of the Open Systems Interconnection (OSI) model and acts based upon user defined criteria. An advanced form of packet filtering called stateful packet filtering exists when the firewall maintains the state of active sessions. The first packet in the session is compared to the filter rules. If the packet is allowed an entry is created in the state table and successive packets belonging to the same session are allowed without having to pass through the rules test.

```
21/04/2001 22:48:52.238630 STATE:NEW 192.168.2.34,137 ->
192.168.2.255,137 PR udp
21/04/2001 22:50:51.960067 STATE:EXPIRE 192.168.2.34,137 ->
192.168.2.255,137 PR udp Pkts 1 Bytes 78
21/04/2001 22:51:18.565050 STATE:NEW 10.10.10.5,1051 ->
192.168.2.254,22 PR tcp
21/04/2001 22:52:03.960073 STATE:EXPIRE 192.168.10.26,27910
-> 10.10.10.1,27900 PR udp Pkts 2 Bytes 774
```

The above is an example of state table entries in OpenBSD. The first and second entries are from a NetBIOS name service broadcast which expired after two minutes of inactivity. The third entry is from a newly established SSH session. The fourth is the expiration of a session from a Quake server advertisement.

Packet filtering which does not maintain state compares every packet to the rules list and acts accordingly. The downside of not keeping state is that every packet must traverse the rules before being blocked or allowed. If your ruleset is large, the load on your firewall can be burdensome.

The advantage of having a stateful packet filter is the added control over what is and is not allowed through the firewall. Since IP header information including sequence numbers and time-to-live (TTL) values are kept in state for active sessions, bogus packets claiming to be from an established session will be denied. A stateless packet filter would not have any basis for denying the packet and the traffic would be allowed.

The other basic type of firewall is the application layer gateway or proxy. A true proxying firewall prevents any packet transfer from one side of the firewall to other. In its most secure form, IP forwarding will not be enabled in the kernel. This is known as a dual-homed configuration. Application proxies reside on the firewall and act as surrogates to the original traffic, accepting source packets from one side of the firewall

and creating new packets to forward on to the destination. This design effectively isolates heterogeneous internal systems and the peculiarities of their TCP/IP stacks from having any contact with external devices.

There is a great deal of flexibility in firewall configuration by utilizing elements of both packet filtering and proxying. Large organizations requiring a highly secure environment but also requiring a great deal of functionality may use a stateful packet filter to block all but specific TCP/UDP ports in combination with an application gateway for all traffic from internal hosts. Smaller businesses may choose to only implement non-stateful packet filtering and no proxy services. While the 'best' configuration depends on the needs of the site, the flexibility in firewall options makes it possible to address most requirements.

The design of an inexpensive firewall solution for a small business must account for the operating system and hardware available for the task. The goal is to keep costs at a minimum while not sacrificing security, stability or reliability. Many organizations have rollover plans to replace aging desktop workstations and servers. Older systems can easily be reapportioned to various 'back end' tasks, one of which might be as a network firewall. For organizations lacking this type of replace and reuse capability, there are a number of outlets for obtaining obsolete hardware at a reasonable cost.

While the types of available hardware may vary, the systems that seem most abundant in many businesses are those of the Intel x86 architecture. Other types may be available, such as the Alpha, Sun or Power PC platforms, however, this paper will focus on the use of dated x86 hardware for the given purpose.

There are a number of open source operating systems that have favorable characteristics for building network firewalls. These characteristics include cost, stability, reliability, performance, and scalability. GNU/Linux [10] is probably the most well known UNIX-like operating system running on the x86 system architecture. Its popularity has resulted in not only a large number of available applications, but also a wide range of support options. These include mailing lists, usenet news, web sites and various commercial support offerings.

GNU/Linux is a stable and reliable platform for firewalling. It is easily configured to support different requirements for secure networking such as network address translation (NAT), port forwarding, and packet filtering. The GNU/Linux kernel just went through a recent upgrade to version 2.4. As a result, numerous

changes were made to the packet-filtering capabilities requiring a complete upgrade and redesign of existing firewalls if use of the new kernel is desired.

Another open source operating system which may be more favorable than GNU/Linux in many implementations is OpenBSD [11]. OpenBSD is based on the original 4.4BSD public source release. During a two year period starting in 1996, the OpenBSD source code went through an intensive line by line audit for potential vulnerabilities. The default build of OpenBSD has been optimized for security making it an ideal candidate for an open source firewall solution. Since it is based on the original 4.4BSD source tree, it is a very mature OS supporting options like stateful packet filtering, large partitions and filesystem journaling which are only now becoming available for Linux. Performance tests [12] also suggest a faster TCP/IP stack and better I/O than Linux 2.2.x on x86.

Since OpenBSD is optimized for security, it natively supports a number of enhanced encryption capabilities such as SSH remote shells, Kerberos authentication and password encryption using algorithms like blowfish, 3DES, and RSA. OpenBSD also touts the fact that in three years there has never been a remote hole in the default install.

Most UNIX-like operating systems support packet filtering though not all support keeping state. Up until the release of the GNU/Linux kernel 2.4, the kernel did not support stateful packet filtering. However, advances in kernel design and the use of the new netfilter application have made stateful packet filtering possible on the Linux platform. OpenBSD, on the other hand, has provided native support for stateful packet filtering since its inception in 1996.

There are other operating systems which could easily meet the requirements outlined above. Among them include FreeBSD and NetBSD. While, after proper hardening, any of these operating systems would be capable of performing the required tasks, each has been developed with specific functionality in mind, (i.e. FreeBSD: performance on i386, NetBSD: portability to different architectures). Length restrictions require that the discussion of other capable platforms be limited. However, the reader is encouraged to investigate other available open source operating systems and evaluate them according to their own needs.

Normally the default install of any operating system is going to have many vulnerabilities which need to be addressed before the system is placed online. That these vulnerabilities are present in the OS has less to

do with laziness on the part of the manufacturer and more to do with the fact that new vulnerabilities are always being discovered and software is constantly being upgraded.

There are a number of resources useful for maintaining the stability and security of any operating system. OS vendors will usually have information on critical security updates and how to apply patches to secure your system. Among the more prominent resources on newly discovered vulnerabilities is Bugtraq [13]. Subscribing to the Bugtraq mailing list is one important way to get timely information on protecting your site. Two other important outlets for information on currently active threats is the Computer Emergency Response Team (CERT) [14] and the System Administration and Network Security Institute (SANS) [15]. SANS compiles the SANS Top Ten which provides information on the most common threats to computer security and how to eliminate them on your site. The use of these resources provides an important way for security personnel to learn about and ensure the security of their sites.

Table 2.1 – Open Source Operating Systems

<i>OS</i>	<i>Advantages</i>	<i>Disadvantages</i>
GNU/Linux v2.2.x	Multiple architectures	No kernel support for keeping state (v2.2.x)
	Wide user support	Numerous vulnerabilities in default install.
	Many compatible applications	
OpenBSD v2.8	'Secure by default'	Not as widely supported
	High performance	Steeper learning curve for inexperienced admins
	OS source code thoroughly audited	

There are many open source proxy applications for individual daemon services. Squid [16] is one example of an application proxy for internal http traffic to external web sites. Squid has many advanced features including configurable caching of http traffic, load balancing over multiple proxy servers, and filtering capabilities. For smaller sites with limited bandwidth, Squid can dramatically increase the performance of web browsing through its caching and read-ahead features.

One firewall proxy suite which encompasses many applications in one package and which is available in source form is the Firewall Tool Kit (FWTK) from Trusted Information Systems [17]. The FWTK is a set of applications which act as proxies for production services which would normally be directly connected to the outside world. Daemon services such as SMTP, FTP, HTTP, as well as generic port forwarding are easily handled at the application layer and passed through the firewall by means of a corresponding

proxy. FWTK is based on the source code from which the original Gauntlet [18] firewall was built, though their source trees are now widely divergent.

FWTK is not under active development and the license restrictions prevent it from being widely implemented and supported by third party support providers. While the source code is available for free, it is tightly controlled by Network Associates, who purchased Trusted Information Systems, and who now produce the Gauntlet firewall product. The license restricts redistribution of the source code and does not allow third party commercial support of the software. In other words, users may download, compile, and use FWTK within their own organization. However, they cannot pay for, nor can someone sell FWTK support.

There have been many efforts to change these restrictions and make FWTK a true open source initiative, so far without success. However, for the dedicated individual that has the desire to make FWTK work in their environment, the user support community is extremely helpful in solving most, if not all, problems dealing with FWTK implementation. FWTK source will compile on a number of operating platforms including Linux and OpenBSD.

Table 2.2 – Application Proxies

<i>Application</i>	<i>Advantages</i>	<i>Disadvantages</i>
Squid HTTP proxy	Integrated caching support	Requires extensive tuning to get best performance
	HTTP filtering capabilities	High put a heavy load on a single server configuration with multiple roles
	Load balancing across multiple servers/networks	
FWTK v2.1	High level of security for included application proxies	Restrictive licensing prevents widespread use
	Many user provided enhancements and patches	No longer actively developed by copyright owners
	Strong user support community	

2.2 System Integrity/Host Intrusion Detection

Host based intrusion detection (HID) involves using tools to detect unauthorized modifications to a specific host. Sometimes referred to as system integrity software, HID detects changes in file size, inode number, permissions, etc. which could indicate undesirable activity occurring on the host. System integrity tools also include the capability of storing MD5, CRC32 and/or SHA1 secure hashes of user defined files when the software is initially installed.

Subsequent integrity checks can verify changes to these files.

Tripwire [19] is a well supported, proven HID system. It is available for a number of different platforms and is scalable to the enterprise. It is also a commercial product which can be very expensive for small companies. However, there are two versions of Tripwire which are available as source code. The academic source release is available to qualified educational institutions and academics. It is not the latest version of the software and does not support some of the newer features such as an encrypted database. However, it is a stable and reliable solution if your needs are limited.

Tripwire for Linux v2.2.1 has been released under the GNU Public License (GPL) [20] and is now being actively developed by the open source community [21]. This version supports database encryption which is lacking from the academic source version. Encrypting the system integrity database protects it from an intruder who might otherwise be able to manipulate it to hide their actions. If the database were not encrypted, it would be possible to programmatically modify the system integrity database during a host compromise. Subsequent integrity checks would not reveal the presence of the compromised system files and hide the actions of the intruder. Database encryption is another mechanism by which Tripwire for Linux ensures the integrity of the host.

Since the GPL'd version of Tripwire is written for Linux, it will not compile under other operating environments like OpenBSD. However, precompiled binaries of the Linux based software can be run in Linux compatibility mode on other operating systems like OpenBSD and FreeBSD. This makes the GPL'd Linux version of Tripwire a popular option for host based intrusion detection.

Another HID system is a program called the Advanced Intrusion Detection Engine or AIDE [22]. It is designed to be very similar to Tripwire in its capabilities, however, like the academic source version of Tripwire, it does not yet support database encryption. Its configuration and policy file are very similar to Tripwire making migration a simple task. AIDE has also been released under the GPL and will compile on many different OS's.

2.3 Central Logging

System logs provide a critical layer in monitoring the security of your network. Because of this, it is often the first thing to be tampered with by a malicious

Table 2.3 – Host Based Intrusion Detection

<i>Application</i>	<i>Advantages</i>	<i>Disadvantages</i>
Tripwire v.2.2.1	Centralized integrity checking for multiple hosts	Cumbersome management
	Recently moved from commercial to GPL license	No GUI based central console
	Native DB encryption	Linux only (Plans for FreeBSD and other ports)
AIDE v0.7	Multiple levels of system integrity checks	New product with few bells and whistles
	Intuitive configuration file	No encryption
	Multi platform support	No site management

intruder trying to cover their tracks. Syslog provides an easy way to send system log messages to a remote server dedicated as a central log repository. By having a replica of all system logs for a given machine you are able to protect yourself from having the original logs modified.

A convenient way to implement centralized logging is by using the remote logging capabilities of syslog. A central log server listens for log messages sent from remote servers. The messages are sent to the central server in plain ascii text using UDP as a transport. However, sending system logs over an untrusted network in plain text makes it very easy for anyone with a packet sniffer to see those logs without having to gain access to either the logging host or the remote server.

One way to alleviate this issue is to send the logs over an encrypted channel using a combination of syslog-ng [23] and stunnel [24]. Stunnel allows you to create an encrypted tunnel between two machines making it possible to send sensitive data over the network in a secure manner. Stunnel uses SSL to build a secure TCP connection on arbitrary ports between two devices. Unfortunately, since syslog uses UDP as a transport it is prevented from functioning with stunnel.

Syslog-ng functions as a replacement for syslog on most UNIX-like operating environments. It has many new features like regular expression matching of system log messages, hashing of log files, and forwarding of logs over TCP connections. Using syslog-ng in combination with stunnel, it is possible to send your logs over an encrypted tunnel to your central log server.

Monitoring system logs is not the most exciting task, especially if you have a central log server collecting logs for many machines on a network. It can be very tedious without an easy and effective way to parse the

Table 2.4 – Centralized Logging

<i>Application</i>	<i>Advantages</i>	<i>Disadvantages</i>
Syslog	Fast delivery of log messages	UDP protocol means unreliable delivery of log messages
Syslog-ng	Standard UNIX software	TCP has added overhead with possible load issues
	Uses TCP for reliable log delivery	
	Encryption capable using third party TCP tunneling	
	Pattern matching logging	

data you want. Luckily, there are a number of tools available to help.

Psionic Logcheck [25] is a package which monitors system log files at regular intervals searching for user specified regular expressions and sending alerts if any of those expressions are matched. The software has a compiled executable which monitors log file changes between specified run intervals, and a shell script which is configured with variables pointing to the log files to be monitored. There are also a few configuration files listing the regular expressions to match against. The regular expressions use a standard syntax and are easily modified for different environments.

Another tool for monitoring log files is a package called Swatch [26] or the Simple Watcher. Swatch has been around for many years and was first introduced at the 1993 LISA conference. It performs some of the functions of Logcheck but is a smaller package and has some additional functionality. Swatch is run once and monitors your logs in real time versus running as a scheduled cron task. This has many advantages in situations where real time notification of security events is absolutely necessary.

Some of Swatch's other capabilities include executing arbitrary code, color coding program output, and piping output to commands, all based on user defined criteria. However, Swatch is limited to monitoring a single logfile at a time, unless multiple instances are run concurrently. This makes it difficult to configure monitoring multiple logs under one process.

2.4 Encryption Protocols

For many years, all network traffic was unencrypted, plain ascii text, easily viewed by anyone running a packet sniffer. Even today, most email, ftp downloads, irc, and web traffic transport their data in plain ascii text. While encryption technology is gradually making its way into mainstream, everyday use, there are definitely programs available now to

Table 2.5 – Logfile Monitoring

<i>Application</i>	<i>Advantages</i>	<i>Disadvantages</i>
Logwatch v.1.1.1	Standard pattern matching syntax	Periodic vs. realtime monitoring
Swatch	Easily configured for multiple log file formats	No active alerting capability
	Realtime monitoring vs. periodic checking	Cannot monitor multiple logs in different locations
	Active alerting and program execution	

allow the use of high encryption for most, if not all, traffic over untrusted networks.

Encryption of network traffic normally occurs in one of two ways. A person uses an application to manually encrypt the data prior to sending it over the network. This is the common method for encrypting messages like email or news posts. The encryption is handled by the user at the application layer. Only the specific data encrypted by the end user is secured. Mail headers and any other information outside the envelope of encryption is left in plain text.

Common programs for encrypting files and messages include Pretty Good Privacy [27] and GNU Privacy Guard [28], both of which use a technology called public key cryptography. It is far beyond the scope of this paper to detail the technical aspects of public key cryptography, suffice to say that this technology is making military grade encryption possible for those wishing to protect their files and data.

Table 2.6 – Encryption

<i>Application</i>	<i>Advantages</i>	<i>Disadvantages</i>
Pretty Good Privacy, PGP	The first widely accessible encryption program	Most development in recent years has been on the Windows client
	Support for multiple algorithms	Now a commercial program with free version available
GNU Privacy Guard	Open Source	Clumsy GUI support
	Backwards compatible with PGP secret keys	
	Wide user support and GUI add-ons	

Another method of encrypting network traffic is by either using an application which supports native transport of encrypted packets such as OpenSSH [29], and OpenSSL [30] or by sending all data and packets through an encrypted tunnel using an application like Stunnel or a protocol like IPSEC [31] and Point-to-Point Tunneling Protocol (PPTP) [32]. Encrypted tunnels are the basis for VPN's. For applications that do not natively support encryption, a tunneling

protocol is an ideal way to encrypt all the traffic between systems on opposite ends of a tunnel.

Table 2.7 – Tunneling Applications/Protocols

Protocol/App	Advantages	Disadvantages
Stunnel	Quick tunnel for specific applications	Works on the transport layer and limited to TCP
IPSEC	Functions on the network layer Highly secure Internet standard	Limited application support
PPTP	Application support on multiple platforms	Known vulnerabilities Not an internet standard Becoming obsolete with advances in IPSEC support
OpenSSH	Terminal access Capable of tunneling arbitrary connections Useful for encrypted file transfer	No UDP encryption
OpenSSL	Create self signed SSL certificates Open Source alternative to Verisign	Self signed certificates are not trusted by remote users Need a root Certificate Authority for production sites

2.5 Network Intrusion Detection

While host based intrusion detection monitors the integrity of individual hosts, this does nothing to detect or monitor malicious traffic on the network. Network based intrusion detection (NID) fills this gap by providing a way to see exactly what packets are traversing your network allowing you to evaluate whether or not they should be permitted. There are two types of NIDS: anomaly detection and pattern matching [33].

Anomaly detection compares network traffic patterns to an established baseline of normal activity. Deviations from that norm are flagged for further analysis. One benefit of anomaly detection is that it does not rely on predefined signatures for detecting abnormal activity. Rather it uses a statistical analysis to determine whether current traffic falls within the parameters of ‘normal’ activity. In this way it can detect previously unrecognized activity or new intrusion methods which are not detectable through other means.

Pattern matching, as the name suggests, compares packets against predefined signatures. The goal is to detect specific traffic based on a given signature and then optionally taking some type of action. Signatures

are established through analysis of previous intrusions. Characteristics which can be used to identify a specific intrusion attempt, such as a particular combination of TCP flags or a TTL value which falls within a certain range, can be used to create a profile of the activity making it detectable through packet analysis. The advantage of pattern matching is that it can be a quick, efficient way to screen for known intrusion activity. However, unlike anomaly detection, it cannot detect unknown attacks.

NID systems are normally either installed directly on a firewall or on a separate system configured to listen on all traffic going through the firewall. Most modern switches provide this capability through port mirroring.

There are a number of freely available NID systems. One popular and highly effective NIDS is Snort [34]. Snort, released under the GNU Public License, is under constant development and is continuously having new capabilities added. Snort relies on user defined rulesets to detect and flag suspect network traffic. The rules language is very flexible allowing pattern matching on packet headers and data content. These sample rules [35] look for a specific virus and backdoor activity.

```
alert tcp any any -> $HOME_NET 139 (msg:"Virus – Possible QAZ
Worm Infection"; flags:A; content: "/71 61 7a 77 73 78 2e 68 73
71/"; reference:MCAFEE,98775;)
alert tcp $EXTERNAL_NET 27374 -> $HOME_NET any (msg:
"BACKDOOR SIG – SubSeven 22"; flags: A+; content:
"|0d0a5b52504c5d3030320d0a|"; reference:arachnids,485;)
```

Newer plugins for Snort enable more advanced features like statistical anomaly detection and fragmentation reassembly. There are a number of logging options a user can choose such as logging directly to syslog, a flat ascii text file, XML, or a number of client server databases including MySQL and Postgres SQL.

Snort was designed to be easily extended through plugins which add new capability to the core program. There has also been a minor explosion in Snort add-ons and partner programs. Some of these work directly with Snort output to create customized reports, correlate attacks, and provide active, real time alerting of intrusion attempts and suspicious network traffic. Most of these add-ons are also GPL'd open source and are freely available.

A support tool useful for analyzing data captured by Snort is the Analysis Console for Intrusion Databases (ACID) [36]. ACID is a web application which interacts with the database tables populated by Snort. ACID has features which allow for close examination

of individual packets and the calculation of summary statistics, useful for observing trends and patterns. Whois lookup capabilities and detailed querying of the packet database make it possible to thoroughly investigate where a packet originated, what it contains, and its context with other network traffic, whether concurrent or not. While there are other ways to extract usable information from Snort logs, the combination of ACID and MySQL is a very convenient and useful option (Figure 2.1).

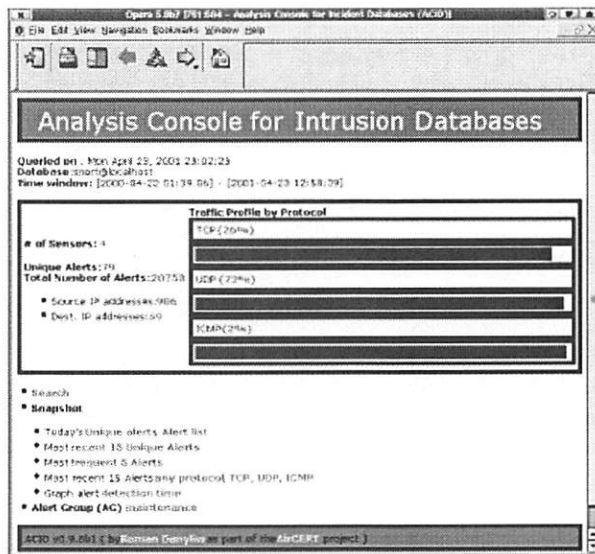


Figure 2.1: ACID Console

Another pseudo NID software is tcpdump [37], included on most UNIX-like OS's. More of a highly configurable packet logger, tcpdump is based on the same packet capture library as Snort and therefore suffers some of the same limitations under heavy load. Tcpdump is a great diagnostic tool but was never intended to perform advanced intrusion detection. However, it is included here as part of an overall combination of applications useful for intrusion detection.

Ethereal [38] is a GUI based protocol analyzer which, like Snort and tcpdump, also uses the pcap packet capture library. Ethereal, while capable of capturing raw traffic, really shines in its usefulness in analyzing tcpdump format packet capture files. Raw packets are shown in both hex and ascii and deciphered into an easily readable format for easy scanning (Figure 2.2).

3. Case Study – Applied Geographics

While it is one thing to describe the free tools available for helping to secure a network it is often helpful to provide an example of them implemented in a real world situation. A case study can be very informative regarding both the problems encountered,

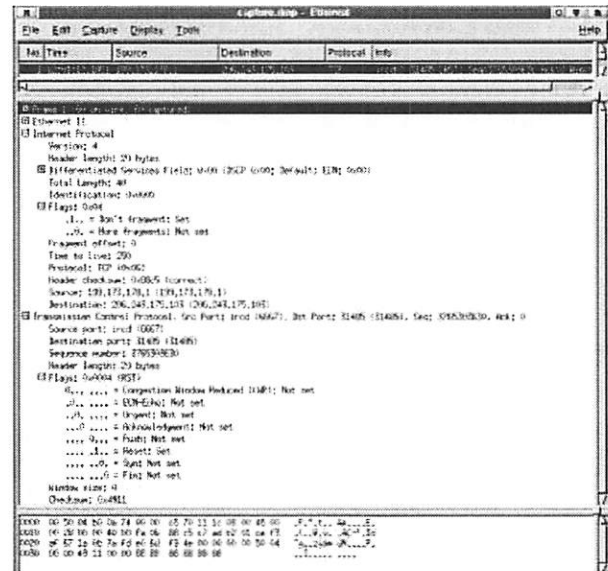


Figure 2.2: Ethereal Console

as well as the benefits, should someone want to replicate the environment for themselves.

Applied Geographics, Inc. (AGI) [39] is a Geographic Information Systems (GIS) consulting company specializing in municipal GIS, web based map server development, and facilities database integration for large organizations. Much of the companies web development projects evolve as prototype map applications running on hosted Microsoft IIS web servers at AGI. These web based applications are made available to clients during the development phase. Since client access is very mobile, it is not possible to restrict access through IP address filtering. Therefore, diligent efforts must be made to protect both AGI's and our clients interests.

Table 2.8 – Network Intrusion Detection

Applications	Advantages	Disadvantages
Snort	Very flexible detection and reporting	Can drop packets under load
	Multiple plugins available	
Tcpdump	Advanced packet logging	Can drop packets under load
Ethereal	GUI interface	Known root exploits
	Excellent interface for analyzing packet logs	

3.1 Past Suffering

In June of 1999, AGI suffered the results of the ExploreZip worm which swept through large segments of the Internet [40]. The worm tore through the companies file servers zeroing out thousands of documents and development project files and generally wreaking havoc on the email system. This

was not an uncommon occurrence for companies relying on Microsoft technology for their email and office productivity software.

If anything positive came out of the ExploreZip event, it was a greater awareness of the threats that exist to companies directly connected to the Internet. When I came to AGI in September of 1999, ExploreZip was still fresh in the minds of the company. It was made clear that protection from the type of threat ExploreZip represented, i.e. opportunistic exploitation of critical system wide vulnerabilities, was a top priority.

I conducted an audit of our existing exposure and found a number of things which needed attention. These included an unpatched Linux 2.0.x firewall with several vulnerable daemons including bind, wu-ftpd, and rpc, no packet filtering, an ftp server open to anonymous uploads, an open mail relay which I later found was being used as a spam relay, telnet access, and intranet fileservers browsing access available from outside the firewall. Any one of these vulnerable points of exposure could have resulted in disaster and it's a wonder that the phone wasn't ringing off the hook with angry system administrators from remote sites.

3.2 Needs Assessment

The first step in designing a security solution is to determine what exactly needs to be secured. This may sound absurd but having a set of guidelines to address specific network security issues is very important. What is the required functionality for internal users? What external access is required to internal resources? Is it necessary to encrypt all network traffic or only external authentication traffic? These are questions which need to be answered before proceeding to installation and configuration of particular software.

At the time of the initial assessment the network environment at AGI was represented by a 40-50 node TCP/IP network running a heterogeneous mixture of Windows NT Server, Windows NT Workstation, Linux, and Digital UNIX. Connectivity to the Internet was through a 416 Kbps SDSL link. Since we were actively replacing our older workstations with newer systems, we had a potentially ready supply of backend, special purpose servers at our disposal.

Required functionality was defined in terms of what services to provide employees on the inside, and what services were required to serve AGI's clients on the outside. The design of the solution also needed to be flexible enough to support future added functionality without sacrificing security or core services. As

always, cost was an important element in the design decisions.

3.3 Internal Services

AGI's network was a relatively open environment with few restrictions on access to resources. As a small company, employees were used to having mostly unfettered access to any and all resources within the corporate network. Therefore, most efforts to protect the internal network were focused on external threats. To minimize potential threats posed by accessing remote resources, access to common internet resources needed to be proxied. Also, network address translation was required for any connections where a proxy was not available.

To accommodate these requirements a new firewall was designed using Red Hat Linux v6.1 [41] with a 2.2.x kernel. While the GNU/Linux kernel v2.2.x does not support stateful packet inspection, it was felt at the time that the trade off of using Linux for its software compatibility versus an OS platform that maintains state was acceptable. Red Hat was readily available and therefore chosen as the firewall OS.

While a default install of most operating systems is quite insecure, it is fairly straightforward to put the system into a secure state through post installation modifications. A spare 100MHz Pentium system with 96MB of RAM was chosen as the hardware platform for the new firewall. A minimal install of Red Hat 6.1 was then hardened by removing all unneeded services, locking down user authentication, installing OpenSSH for terminal access, and creating ACL's for access to both local and remote services.

Since the main concern was for external security there was no requirement to encrypt the transmission of syslog messages to the central server. A central log server was installed and configured to use syslog as a listener for the internal network. Syslog was also a requirement due to the heterogeneous nature of AGI's network. Syslog-ng is not supported on older Digital UNIX and Windows NT systems, all of which needed to forward their log messages to the central server.

Since the amount of traffic passing through the firewall was relatively minimal the packet filter and application gateway were configured on the same device. In larger implementations, this may not be appropriate.

Squid was installed to provide gateway services to all external http resources. The added functionality and performance of Squid made it an appropriate choice for AGI. Since http traffic is the primary external

resource accessed at AGI, it was felt to be important to have a proxy in place. Many other services, including Real Audio and ICQ, have the ability to use http as a transport obviating the need for additional proxy support beyond Squid. Other resources, such as external ftp servers, were kept unproxied.

Network address translation (NAT) hides internal devices that access external resources by replacing the source ip address of the internal system with the ip address of the firewall. The ipchains [42] utility allows for the easy configuration of NAT using one simple rule:

```
# ipchains -A forward -i SEXT -j MASQ
```

This will masquerade all packets exiting the local network on the external interface.

3.4 External Services

Providing access to services and resources from external, untrusted networks requires careful planning to minimize threat exposure. It was necessary for AGI to provide a number of services to clients, employees, and the general public. These services consisted of an SMTP mail server for local and remote mail delivery, web hosting for both AGI and our client prototype applications, an ftp server for anonymous downloads and authenticated uploads, a VPN for telecommuting, and an administrative console capability.

The following action items were established to satisfy the functional requirements of the AGI network. First, a packet filter was needed to provide access controls to the AGI network. These included egress filtering of RFC 1597 private addresses. While this step does not necessarily protect AGI, it does promote responsible network management. Second, daemon services should be proxied where possible. Third, AGI employees should be able to access internal network resources from home or on the road. Fourth, at a minimum, all authentication of external sessions to internal resources (web access, VPN, terminal) should be encrypted. If possible, the entire session should also be encrypted.

The v2.2.x Linux kernel uses a program called ipchains to manage packet filtering. The three default chains, input, output, and forward, were all defaulted to deny access. By denying everything and then selectively allowing specific traffic, you create an awareness of the nature of all traffic entering or leaving the protected site. You also protect yourself from overlooking a type of traffic that would normally not be allowed. At AGI, it was necessary to allow access to a number of specific TCP and UDP ports

including 21 (FTP), 22 (SSH), 25 (SMTP), 80 (HTTP), 143 (IMAP), 443 (HTTPS) and 993 (IMAPS). Servers listening on these ports were regularly monitored and patched to prevent possible compromise.

Egress filtering is a method to prevent broken packets or packets with non-routable addresses from being routed outside the protected network. All packets with RFC 1918 source addresses (10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16) should be blocked from routing to the internet. This is good networking practice and reduces the routing of unwanted traffic.

The TIS Firewall Toolkit (FWTK) provided the application gateway support required for SMTP transport. The original source code for FWTK's smap proxy does not support a restricted mail relay or spam filtering. The end result is that while sites are protected from exposing their mail servers directly to the internet, the proxy, itself, is vulnerable as an open spam relay. However, there are a number of highly effective patches to the original FWTK source code which add the necessary open relay controls while also providing support for spam filtering. This makes the FWTK smap proxy a usable mail proxy for AGI's site. The FWTK also supports proxies for a number of other application services such as FTP, HTTP, X server, Telnet, and SMTP. However, the SMTP gateway was the only FWTK server proxy used at AGI.

In addition to filtering and proxying network traffic to and from AGI, it was necessary to encrypt as much of the remaining traffic as possible, protecting usernames, passwords and sensitive data. E-mail encryption is still fairly non-standard and cumbersome making it difficult to implement in any mildly heterogeneous environment, let alone with any outside clients and contacts. These unresolved issues made it exceedingly difficult to control the use of application level encryption. However, it was possible to control the encryption of user logon sessions to IMAP mailboxes, access to internal web based information, and remote access.

The primary concern was authenticated, encrypted access by AGI employee's and clients to protected resources. Creating a root Certificate Authority (CA) for AGI permitted the creation of self-signed web and email server SSL certificates. These certificates would permit SSL access for testing and pre-production phases of the development cycle. OpenSSL was used to create a root certificate which was then used to sign certificates created by web and email servers. Most modern web and email servers including IIS, Apache and Exchange, are capable of

utilizing SSL transport of data. Once the certificates were installed on the servers, users of the site needed to install the CA root certificate as a trusted root authority in the web browser. This prevented errors about untrusted root certificates from being generated by the browser. Email clients were configured to use port 993 (IMAP over SSL) to connect to the IMAP server. These two steps eliminated 90% of the plaintext authentication and data stream to/from AGI's protected resources.

One of the core requirements for a new firewall solution was an effective remote access capability that did not open AGI to excessive risk. Any solution needed to rely on encryption of initial session negotiation, authentication, and data flow.

As a GIS solutions provider, there are specific software dependencies that AGI must support. In the past this software was multi-platform allowing the use of any number of operating system and hardware choices. AGI's legacy GIS applications run on aging Digital UNIX servers accessed through X terminal sessions. These applications and hardware platforms continue to be supported at AGI. However, changes in the GIS software market have dictated a migration to a Microsoft-centric operating platform at the desktop. In addition, widely available broadband internet connections have made telecommuting a viable alternative to battling the morning commute for many of AGI employee's. Initial research into a solution enumerated a couple of potential open source solutions such as PoPToP [43] and FreeS/WAN [44]. However, both were fairly new initiatives which did not fully support the type of access required, vis à vis full replication of the desktop environment at AGI through an encrypted tunnel. The decision was made to use PPTP on NT Server passing the traffic through the firewall.

Since the focus of this paper is on the use of free, open source security solutions, a discussion of the relative merits and limitations of using Microsoft's PPTP implementation versus some of the more reliable, secure or open source VPN solutions on the market will be avoided. Given further refinement of the existing initiatives, an open source solution would provide an attractive alternative. However, while most open source VPN solutions focus on providing secure router <=> router links using IPSEC thereby obviating the need for dedicated client software, any VPN remote access solution for workstation <=> server would require a software client developed for the Windows desktop. Commercial products are available which fill this gap, however, open source VPN client software that supports IPsec and is able to run on the Windows desktop is noticeably absent.

In addition to a secure remote access solution for AGI employees, a secure method of administering the network while off site was also required. OpenSSH provided a highly reliable and secure method for establishing terminal connections to AGI network servers from remote sites. OpenSSH has many configurable parameters including key length, cipher, and authentication mechanisms such as Kerberos, RSA key authentication and username/password combinations. OpenSSH also provides methods for creating encrypted tunnels over arbitrary TCP ports, secure file transmissions, and secure X window negotiation if desired.

3.5 System Integrity

One of the key problems with host compromises is that they are usually not discovered until long after the compromise took place and the intruder has already damaged your site or someone else's. Recent attacks on exploitable vulnerabilities in bind, rpc, wu-ftpd, and lpd make it abundantly clear that undiscovered system compromises contribute to the spread of worms, both kiddies and automated, looking for further victims. Unchecked, these compromised systems pose a serious risk to the stability of the internet.

A good host level intrusion detection capability is a necessary component of any system exposed to direct connections from the internet. At AGI, it was necessary to provide host based intrusion detection on the GNU/Linux firewall as well as any other Linux systems installed on the network. Since the goal was to use open source software, system integrity software for other OS platforms will not be presented.

Two open source solutions were previously described. While both Tripwire and the Advanced Intrusion Detection Engine (AIDE) provide system integrity checking, only the recently open sourced Tripwire 2.2.x for GNU/Linux has built-in capability of encrypting the system database. This functionality is planned for AIDE, however, with the GPL'ing of the Linux client, the maturity of the application, and with the support for running Linux binaries on BSD systems in compatibility mode, Tripwire provided an ideal solution at AGI.

3.6 Network Intrusion Detection

Solutions for an adequate firewall and host level intrusion detection provide a solid base for protecting private networks. However, these tools do not actually allow you to see the traffic entering and leaving the network. A network based intrusion detection system

provides a means to identify unwanted network traffic and take appropriate action.

AGI required a system which could be modified to detect new and customized signatures, log alerts to a central database, and provide the means to analyze detects. Snort provided the ability to create advanced rulesets for detecting unwanted network activity, and report that activity to a remote database. The Snort configuration at AGI combines a subset of rules made available by the Snort development team and a custom ruleset created to account for known traffic patterns on AGI's network. Alerts are logged to a local binary tcpdump format logfile in addition to a remote MySQL database. Analysis of all alerts logged to the MySQL database is conducted using ACID.

In addition to the use of ACID for analyzing the Snort alert database, Ethereal's filtering and sorting capabilities make it an extremely useful companion for analyzing the tcpdump format binary files created by Snort. The drill down capability of Ethereal allows for a complete and precise view of each packet.

The use of these tools highlights another important benefit to using open source software for network security. There are no limitations to the combinations of software that can be used for any specific objective. Commercial software gets expensive if you have to keep buying new packages for additional functionality. If you don't like an open source application, there's no financial incentive to try and 'make' it work if it simply will not serve your purpose. If an application has some useful features but does not provide all the functionality required, there are other packages available which can either replace the existing software or compliment it with the new features. Either way, the investment in the software costs you nothing.

3.7 Cost Considerations

As stated in the introduction, the cost of network security is part and parcel of the cost of doing business online. Hardware, software, time and skills, all come at a price. Formal training and certification programs can be expensive. Conflicting priorities and responsibilities can make it difficult for system administrators to spend the time needed to learn new skills and keep current on the technology. Fortunately, with the availability of inexpensive legacy hardware, free and open source software, and a wealth of information online, those costs have neither to be exorbitant nor unmanageable. In fact, the money spent on network security now may be an investment in your businesses online future.

The choices made at AGI on software and hardware paid off immensely. Through the use of spare hardware, a solid open source OS, and the open source security tools described above, it was possible to implement a highly secure network environment at AGI. The software required to build the solution cost nothing. The cost incurred in implementing the solution was the time spent initially building the system, and the subsequent hours spent updating, monitoring, and maintaining the software. Since these duties fall within the role of System Administrator at AGI, the costs were negligible as a separate line item and were well justified.

3.8 Future Directions

In the year and a half since the original solution was implemented, the GNU/Linux box running on a five year old Pentium 100 desktop workstation, has never crashed, failed or been compromised. It was rebooted twice, due to physical location changes and handled all the network traffic for 35 employees, six hosted production web sites, 6 staging and developmental beta web sites, and up to 10 concurrent VPN connections without any interruption of service.

Recent changes in the network infrastructure at AGI, demands for additional bandwidth, and additional hosting requirements, required moving the firewall to a larger box and adding stateful packet filtering capabilities. Since stateful packet inspection is a recent addition to GNU/Linux and may have some lingering unresolved vulnerabilities [45], and since software compatibility issues are no longer an issue in using *BSD based systems, OpenBSD was used rather than Linux. The move will allow AGI to maintain existing capabilities, while enabling stateful packet filtering for better performance and less overhead.

The bandwidth at AGI has moved from 416Kbps SDSL to two full 1.544Mbps T1's. The network performance statistics on *BSD vs. Linux [12] as well as the reputation of OpenBSD with its focus on security made the decision to switch an easy one. The capabilities and reliability of open source operating systems and software make them an extremely attractive alternative to commercial OS's and security software.

4. Summary

In the mid-1990's, when computer programmers and system integrators realized that a date change from the year 1999 to 2000 might have major implications for government and business computer systems, billions of dollars were spent resolving the problem. After literally millions of person hours spent by developers

and technicians repairing code and fixing systems Y2K fizzled on New Years, 2000. Some argued that the money was ill spent since nothing happened. Their logic does not account for the diligent efforts of those who worked on Y2K's demise. It rather suggests that "nothing was going to happen anyway so why did we spend all the money?"

Network security suffers from similar misconceptions. If you take steps to minimize your exposure and suffer no intrusions you are doing your job. But then was all the money and effort really worth the expense? One need only refer to CERT, SANS, NIPC, or a number of other resources specifically designed to keep users informed on the threats that exist.

As mentioned above, the solution implemented at Applied Geographics has never crashed, failed, or been compromised. However, this is not to say that people haven't tried. Since April, 2000, the monitoring of AGI's network resulted in over 30,000 events, including numerous worm detections, root exploit attempts, errant packets and useful traffic analysis leading to network design changes and improvements in overall system performance.

While cost is always an important factor in deciding how best to secure a network, this paper has attempted to show that one need not spend thousands of dollars on commercial software. Any network security solution is going to cost money in the form of someone to provide support, installation and monitoring. However, to believe that you must also incur the added cost of the software is to overlook the solid, cutting edge, and secure open source solutions that are available for free.

5. Acknowledgements

I would like to thank Ted Faber and Peter Girard for their extremely helpful comments and suggestions. Any errors in content are, of course, solely the responsibility of the author.

6. References

- [1] Computer Emergency Response Team. CERT Advisory CA-1999-06. <http://www.cert.org/advisories/CA-1999-06.html>, June, 1999.
- [2] Computer Emergency Response Team. CERT Advisory CA-1999-04. <http://www.cert.org/advisories/CA-1999-04.html>, April, 1999.
- [3] Ohlson, K. "Viruses, Other Attacks Cost Businesses \$7.6B: Report". http://www.computerworld.com/cwi/story/0,1199,NAV47_STO28244,00.html, June, 1999.
- [4] National Infrastructure Protection Center. Alert 00-0034. <http://www.nipc.gov/warnings/alerts/2000/00-034.htm>, February, 2000.
- [5] Vision, M. Ramen Internet Worm Analysis. <http://projet7.tuxfamily.org/docs/security/ramen.html>, 2001.
- [6] System Administration, Networking and Security Institute (SANS) Global Incident Analysis Center (GIAC). Lion Worm. <http://www.sans.org/y2k/lion.htm>, March, 2001.
- [7] SANS GIAC. Adore Worm. <http://www.sans.org/y2k/adore.htm>, April, 2001.
- [8] Strom, D. "The Packet Filter: A Basic Network Security Tool". http://www.sans.org/infosecFAQ/firewall/packet_filter.htm, September, 2000.
- [9] Curtin, M. and M. J. Ranum. Internet Firewalls: Frequently Asked Questions, rev 10.0. <http://www.interhack.net/pubs/fwfaq>, December, 2000.
- [10] GNU/Linux, v2.4.x. <http://www.kernel.org>
- [11] OpenBSD, v2.8. <http://www.openbsd.org>
- [12] Graichen, T. "Performance Comparison and Tuning of Free Operating Systems". <http://innominate.org/~tgr/slides/performance>, 2000.
- [13] Bugtraq vulnerability mailing archive, SecurityFocus.com. <http://www.securityfocus.com/bugtraq/archive>
- [14] Computer Emergency Response Team (CERT). <http://www.cert.org>
- [15] System Administration, Networking and Security Institute (SANS) Global Incident Analysis Center (GIAC). <http://www.sans.org>
- [16] Squid Web Proxy Cache. <http://www.squid-cache.org>

- [17] Firewall Tool Kit, Trusted Information Systems, Network Associates, Inc.
<http://www.fwtk.org/main.html>
- [18] Gauntlet Firewall, PGP Security, Network Associates, Inc.
<http://www.pgp.com/products/gauntlet>
- [19] Tripwire commercial software.
<http://www.tripwire.com>
- [20] GNU General Public License. The GNU Project.
<http://www.gnu.org/philosophy/license-list.html>
- [21] GPL Tripwire project, v2.2.1 for Linux.
<http://sourceforge.net/projects/tripwire>
- [22] Advanced Intrusion Detection Engine (AIDE) v0.70. <http://www.cs.tut.fi/~rammer/aide.html>
- [23] Syslog-ng, v1.4x.
<http://lists.balabit.hu/products/syslog-ng>
- [24] Stunnel Universal SSL Wrapper, v3.14.
<http://www.stunnel.org>
- [25] Logcheck, v1.1.1.
<http://www.psionic.com/abacus/logcheck>
- [26] Hansen, S.E. and E.T. Atkins. "Centralized System Monitoring with Swatch". 1993 LISA Conference. Usenix Association.
<http://www.stanford.edu/~atkins/swatch/lisa93.html>
- [27] Pretty Good Privacy (PGP), PGP Security, Network Associates, Inc. <http://www.pgp.com>
- [28] GNU Privacy Guard, v1.04.
<http://www.gnupg.org>
- [29] OpenSSH, v2.52. <http://www.openssh.org>
- [30] OpenSSL, v0.96a. <http://www.openssl.org>
- [31] Kent, S., and R. Atkinson. "Security Architecture for the Internet Protocol". RFC 2401. November, 1998.
- [32] Hamzeh, K., et. al. "Point-to-Point Tunneling Protocol". RFC 2637. July, 1999.
- [33] Lee, W., C.T. Park, and S.J. Stolfo. "Automated Intrusion Detection Using NFR: Methods and Experiences". In Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings, pages 63–72, April, 1999. Usenix Association. Berkeley, CA.
- [34] Snort Lightweight Intrusion Detection for Networks, v1.7x. <http://www.snort.org>
- [35] Forster, J. Snort Ruleset Database.
<http://www.snort.org/Database/rules.asp>
- [36] Analysis Console for Intrusion Detection (ACID), v0.9.6x. <http://www.cert.org/kb/acid/>
- [37] Tcpdump/Libpcap. <http://www.tcpdump.org>
- [38] Etherreal, v0.8.17. <http://www.ethereal.com>
- [39] Applied Geographics, Inc.
<http://www.appgeo.com>
- [40] ExploreZip.worm.
http://vil.nai.com/vil/dispVirus.asp?virus_k=10339
- [41] Red Hat Linux 6.x. <Http://www.redhat.com>
- [42] Linux IP Firewalling Chains, v1.3.10.
<http://netfilter.filewatcher.org/ipchains/>
- [43] PoPToP, v1.0.1, The PPTP Server for Linux.
<http://poptop.lineo.com>
- [44] Linux FreeS/WAN, v1.9.
<http://www.freeswan.org>
- [45] IPTables FTP Stateful Inspection Arbitrary Filter Rule Insertion Vulnerability. Bugtraq ID #2602.
<http://www.securityfocus.com/bugtraq/archive>

Heimdal and Windows 2000 Kerberos — how to get them to play together

Assar Westerlund
Swedish Institute of Computer Science
assar@sics.se
Johan Danielsson
Center for Parallel Computers, KTH
joda@pdc.kth.se

Abstract

As a practical means of achieving better security and single sign-on, the Kerberos network authentication system has been in wide use in the Unix world for many years.

Microsoft has included its own implementation in Windows 2000, replacing the NTLM authentication system from older Windows NT versions. This facilitates sharing account information between Unix and Windows machines, as there is no need to keep different passwords.

Although Microsoft's Kerberos implementation mostly follows the specification, there are a number of deviations and extensions, not all of which are well documented. Consequently, it is not always obvious how to fit Windows 2000 clients and servers into an existing Kerberos environment. In this paper we discuss the differences between the two systems and describe how we got our Kerberos implementation, Heimdal, to work with Windows 2000.

1 Introduction

Ever since Microsoft announced that Windows NT 5 (later renamed to Windows 2000) would be using Kerberos for network authentication, there have been questions as to how that implementation would interoperate with existing implementations. Considering Microsoft's bad reputation of "embracing and extending" other systems, people feared that what eventually came out would be something that would at best be similar to Kerberos. As it turns out, these fears are mostly unfounded.

While it mostly follows the specification, the Kerberos in Windows 2000 has some small implementation differences and undocumented extensions to the protocol. This makes writing a replacement for the Windows 2000

Kerberos server hard. However, we feel that there are good reasons for using a Windows 2000 Kerberos server to support Windows clients, so this might not be a big problem.

Heimdal[1] is an implementation of Kerberos 5 that we have been working on for some time. To make it work better with Windows 2000, we have made a number of changes. These include adding RC4 encryption, configurable salting of keys (which is required by some other systems as well), and crude support for referrals.

This paper starts with an introduction to the relevant Kerberos concepts in section 2. Section 3 explains the difference between database organisations. Section 4 discusses the different issues that come up when trying to interoperate between Heimdal and the Windows 2000 Kerberos. Section 5 explores different scenarios on how the two systems can be integrated, and finally conclusions and future work are presented in sections 6 and 7.

2 Kerberos

Kerberos is a network security system for authentication. It allows users and services, collectively called *principals*, to authenticate to each other over an insecure network.

Kerberos relies on a central server (the *Kerberos server*) which is trusted by all principals. Starting with this trust relationship, the Kerberos server can securely introduce the communicating parties to each other. The Kerberos server is also called the Key Distribution Centre (KDC). All principals have a secret password or key that they share with the Kerberos server. This allows them to verify that they communicate with the correct Kerberos server, since no other entity should know their password or key.

Although each user has a secret password, the passwords are actually stored as encryption keys in the database. These keys are derived from the passwords with one-way functions (*string-to-key* functions). For services, the keys are stored in a location (typically in a file) where the server program can access them.

A client authenticates to a server by providing the server with a piece of data (the *ticket*) generated by the KDC and encrypted in the server's key. This ticket proves the client's identity to the server. The server may also prove its identity to the client by showing that it can decrypt the ticket. Each ticket contains a session key (also sent to the client) which allows the client and server to encrypt their traffic.

Single sign-on is achieved in Kerberos by using a special *ticket-granting ticket*, that is obtained when a user logs in. This ticket can later be used to get more tickets from the KDC, without having to enter the password again. The ticket identifies the holder as a particular user, so anyone with access to the ticket can impersonate that user. To lessen the damage if a ticket is stolen the ticket has a limited lifetime.

The Kerberos world is divided into *realms*, where each realm is an administrative domain. A realm's name will normally be the same as the site's DNS domain name. The name of a principal is a list of strings, separated by slashes, followed by the realm name. A typical user would be named *nisse@FOO.SE* and a service *host/bar.foo.se@FOO.SE*.

This paper discusses Kerberos 5, the current protocol version. Version 4 was the first to be publicly available and had a sizable installed base when version 5 reached maturity. There are still version 4 based applications and clients in use. Thus, most current version 5 implementations have functionality for handling version 4 clients. Version 5 is reasonably similar to version 4, except that it is more parameterised, including support of several types of encryption algorithms.

Kerberos is described in more detail in [2, 3, 4, 5].

3 Kerberos databases

Every key that the Kerberos server keeps must be stored in some kind of database. The database needs to contain at least the names and keys of the principals. Additional information stored and the organisation of the database can vary quite a lot between different implementations.

3.1 Heimdal's database

On traditional Unix systems, account and password information is stored in a local database (such as */etc/passwd*) or some distributed database (such as NIS).

With a typical Heimdal setup the key database is separate from the account database and password information is not directly available to the clients.

This means that the name space for Unix users and Kerberos principals *can* be different, though normally they are not. Users might have several principals for different roles. For example a user might authenticate as the principal *nisse/root* when acting as super-user and as the principal *nisse/admin* when doing administrative functions with the database. Services have principals in the Kerberos database but not necessarily any corresponding Unix accounts. Even if they do, there is not necessarily a one-to-one mapping between principals and accounts. Services' principals are usually named *service/hostname*. The basic fields of a database entry are shown in Figure 1.

Field	Type
Principal name	list of strings
Principal expiration	date
Password expiration	date
Attributes	flags
Key version	integer
Keys	(encryption type, salt, key)...

Figure 1: A basic Heimdal database entry

There are several different string-to-key functions, so what particular function was used for a key has to be stored along that key. Some functions also take a known string as input, known as *salt*. The reason for the salt is to make comparing keys and performing dictionary attacks harder (if the same password is used in different realms, the resulting keys will not be identical).

The typical way a realm is set up is with one master server where all modifications to the database are performed, and a number of slaves that maintain read-only copies of the same database. Changes are propagated either periodically or incrementally from the master to the slaves. This is similar to the common DNS server configuration with one primary name server and zero or more secondary name servers.

3.2 Windows 2000's database

Windows 2000 uses a data repository called the Active Directory[6] for most of the domain data. This includes the users and machines, and their keys.

The active directory is a hierarchical directory service which stores different kinds of data, each identified by a particular schema. It is distributed among the domain controllers of a domain with multi-master replication. Thus, changes made to any of the servers will be propagated to the other servers.

4 Protocol and implementation issues

4.1 Encryption types

The original Kerberos protocol specification (RFC1510[5]) made DES the required encryption type to implement. Windows 2000 implements this encryption method, and it interoperates with other Kerberos implementations.

When upgrading an NT 4 domain to Windows 2000, there are only MD4 keys for all users, so there is no way to use DES. To support this common case, Microsoft included its own RC4 based encryption algorithm (rc4-hmac-md5) that make use of the MD4 keys. This algorithm is described in a series of drafts[7] published by Microsoft. Heimdal also has an implementation of it, which we have tested against Windows 2000.

4.2 Salting

Normally keys are salted with the principal name, but there are situations when a different salt is used. One example is when converting an existing Kerberos 4 realm to Kerberos 5. In Kerberos 4, the keys are not salted (the salt string is empty). Another is when a principal is renamed, since the principal name will change, but the key will remain the same.

When the salt is non-standard, it has to be stored in the database, and sent to the client. Windows 2000 can do this, but for unknown reasons it does not handle the empty salt.

The Heimdal key database can keep several keys with different salting information (both type and string). The point at which more keys can be added is when the user's password is changed, so there is configuration support for specifying what types of keys should be created

whenever a password is changed.

4.3 Limitations and problems

Windows 2000 does not implement all of the functionality required by the Kerberos specification. One of the required checksum types is not actually implemented (rsa-md5-des). This is a problem because there is no negotiation or possible way of knowing this beforehand.

When a user's password has expired, the Kerberos server will return an error and only allow the user to change their password. Windows 2000 erroneously gave the same error when the user was actually trying to change the password which resulted in an infinite loop in our client. This bug has been fixed in Service Pack 1.

Unfortunately Windows 2000 does not support looking up KDC information for non-2000 realms using DNS, therefore, configuration information has to be added manually.

4.4 Authorisation data (or PAC)

The Kerberos protocol only provides authentication, it proves the identity of a communicating party, and not authorisation, or telling what rights and privileges they might have. The common way of implementing authorisation is to look up the identity in a separate list or database and see what they are authorised for in this context. Microsoft instead tried adding this to Kerberos.

The Windows 2000 KDC adds extra authorisation data to the tickets it generates. This data is called the Privilege Access Certificate (PAC). It includes some information about the user and group memberships. Both users and groups are represented by their Security IDs (SIDs), which is a unique number for every Windows 2000 object. All of this information is stored in the active directory, and the application server should be able to look it up from the client name in the ticket, instead of getting it from the PAC. However, it is unknown whether servers will do that if they get tickets without the PAC.

The PAC data format has been partially reverse-engineered. We wrote code as part of Heimdal to dump the authorisation data and then were assisted by people with much more familiarity with NT data structures. The format has also been documented in a Microsoft document[8] that has a *trade secret* license that prohibits anyone from implementing it.

4.5 Applications

Without applications that use Kerberos, a working infrastructure is not very useful. However, the traditional Kerberos applications on Unix (such as telnet, rsh, and ftp) are not available on Windows 2000 or do not support Kerberos. Applications on Windows 2000 use Kerberos for a number of different protocols such as LDAP, SMB, and COM. Unix counterparts of these applications are only somewhat available.

4.6 Administration

There is no standardised protocol for administering a Kerberos database. Windows 2000 uses the Active Directory to store the database which can be accessed through Kerberised LDAP. Consequently there is a possibility of using non-Microsoft tools to maintain the database information.

Password changes by users are done with a different protocol[9] and it works fine between Heimdal and Windows 2000.

Database propagation between the Kerberos servers is also rather different. Making the active directory distribution and the different propagation methods work together is non-trivial. This makes mixed realms with both Windows 2000 and other servers quite unlikely.

4.7 Referrals

When getting tickets the client has to know what principals to request them for. The traditional way of doing this is to use the user's login-name and the client machine's pre-configured realm name. Microsoft has proposed a draft[10] to extend this mechanism, so it would, for instance, be possible to use an e-mail address as login name. The extension requires some changes to the protocol, since the KDC is not allowed to return a ticket for a different principal than requested.

Referrals can also be used to remove the need for host name lookups on the client, somewhat like turning the KDC into a secure DNS server.

It is unclear how much of this draft has been implemented in any released Windows version. It is a fact that a Windows client will only talk to KDCs in the realm it currently belongs to, unless it gets a referral to another KDC. Thus the KDC needs to have some support for referrals or cross-realm authentication will not work.

We have added functionality for referrals to the Heimdal KDC that is sufficient for Windows clients.

4.8 APIs

Windows 2000 does not support any of the traditional Kerberos 5 library functions, that many Unix applications use.

The GSS-API[11] protocol with a Kerberos mechanism is implemented, but not the API part of it. Kerberos application programming under Windows 2000 is done with the Security Service Provider Interface (SSPI) which is quite similar to GSS-API. Thus only a small effort is required to write code that works with both Windows 2000 and other Kerberos implementations. Because Windows 2000 implements the GSS-API protocol, applications written against SSPI will interoperate with GSS-API applications using other Kerberos implementations.

5 Scenarios

There are different ways to integrate a Windows 2000-based infrastructure with other Kerberos realms. Some of these are discussed here and the interoperability of each of them is explained. Some more details on the exact commands to run are available in [12].

5.1 A Windows 2000 client in a non-2000 realm

A standalone Windows 2000 workstation (a member of a workgroup but not of a domain), can be configured to use a non-2000 realm for login authentication. The ksetup program (which is unfortunately not installed by default but supplied on the Windows 2000 install CD) can be used to configure what realms should be used by a particular workstation. The DNS names of the KDCs also have to be configured (see 4.3).

The workstation must have a key in the Kerberos database. Also the mapping of Kerberos principals to local users (typically one-to-one) has to be configured. It is worth noting that the workstation will use the configured KDC for all its requests, independent of what realms the application servers belong to, so this KDC has to be able to handle these requests (see 4.7). If the configured KDC handles these requests, the workstation can connect to remote Windows 2000 domains.

Sites with a small number of Windows 2000 machines

most likely want to use this configuration. It is being used mainly by sites that do not want to have a Windows 2000 domain or do not want all of their machines to be members of their 2000 domain.

5.2 A non-2000 client in a Windows 2000 realm

Clients using other Kerberos implementations should not need very many changes to interact with a Windows 2000 KDC. Key installation is of course different. The client must have a user in the active directory, created with the normal Windows 2000 tools. Then a mapping between this user and the principal name (*host/fully.qualified.hostname*) has to be installed with the `ktpass` command. The key that resides in the active directory also has to be copied to a file on the client.

We are not aware of anyone using this configuration.

5.3 A Windows 2000 domain with a non-2000 KDC

The problem using a non-Microsoft KDC for a Windows 2000 domain is that the KDC is very much integrated with the other domain controller servers. All of these servers would have to be replaced at the same time, much as Samba [14] can act as a domain controller for an NT 4 domain. It is unclear how much work is needed to make Samba a replacement for the Windows 2000 domain servers but it is probably a large amount. And, of course, the Heimdal KDC would also have to be integrated.

The PAC data also make things more complicated. If there are no native 2000 servers being run in the domain, PACs will not be needed. And it has yet to be determined if native 2000 servers use PACs as optimisations or if they are actually required. The worst case would be having to reverse engineer the complete format and then sew the KDC and the domain controller together.

Getting this working reliably is still far in the future.

5.4 Inter-connected Windows 2000 and non-2000 realms

A Windows 2000 realm can be integrated with an existing non-2000 realm by allowing clients in the Windows 2000 realm to authenticate to the existing realm. When a client wants to authenticate to a server in a different realm, the two realms must share a key, either directly

or indirectly through other realms. Windows 2000 and non-2000 realms can share a key. All the involved Windows 2000 clients need to have configuration information about the foreign KDCs. Once the foreign KDC information is stored on the domain controller of the Windows 2000 domain, a key can be configured with the GUI administrative tool. This key also needs to be added to the other realm. Kerberos authentication can then take place between the two realms. The domain can also be configured with the same GUI tool to allow users to login when they are authentication in the other realm, similar to configuration of a standalone workstation.

This is likely to be the most common configuration, since there is only one password database and all Windows applications that rely on or make use of the domain infrastructure still work. There are several large sites that have their realms set up this way, keeping all their users in one realm and all the Windows machines in a Windows realm.

6 Conclusions

While Windows 2000 Kerberos is different, getting it to work with other Kerberos implementations is not that hard. The documentation is not always sufficient, and sometimes experiments have to be performed to figure out how things actually work.

7 Future work

Windows 2000 uses an extension for using public key cryptography in initial authentication[13]. An implementation of this in Heimdal would be useful, not only for use with Windows.

Assuming that there is a usable specification of the PAC format (see 4.4), integrating the Heimdal KDC with Samba[14] to create an entire domain controller would be useful.

8 Availability

Heimdal is freely available from
<http://www.pdc.kth.se/heimdal/>.

References

- [1] Assar Westerlund and Johan Danielsson, *Heimdal — an independent implementation of Kerberos 5*,

- In proceedings of the Usenix 1998 Annual Technical Conference, New Orleans, USA, June 1998, Freenix Track
- [2] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, Proceedings Winter USENIX Conference, Dallas (1988)
- [3] John T. Kohl, B. Clifford Neuman, and Theodore Y. T'so, *The Evolution of the Kerberos Authentication System*, Distributed Open Systems, pages 78-94. IEEE Computer Society Press (1994).
- [4] B. Clifford Neuman, and Theodore Ts'o, *Kerberos: An Authentication Service for Computer Networks*, IEEE Communications, 32(9), pages 33-38 (1994)
- [5] John Kohl and Clifford Neuman, *The Kerberos Network Authentication Service (V5)*, Network Working Group (1993), RFC1510
- [6] Microsoft, *Directory Services*, <http://www.microsoft.com/windows2000/library/technologies/activedirectory/default.asp>
- [7] M. Swift and J. Brezak, *The Windows 2000 RC4-HMAC Kerberos encryption type*, Work In Progress, draft-brezak-win2k-krb-rc4-hmac-02.txt
- [8] Microsoft, *Kerberos PAC specification*, <http://www.microsoft.com/technet/security/kerberos/default.asp>
- [9] M. Horowitz, *Kerberos Change Password Protocol*, Work In Progress, draft-ietf-cat-kerb-chg-password-02.txt
- [10] M. Swift, J. Brezak, J. Trostle, and K. Raeburn, *Generating KDC Referrals to locate Kerberos realms*, Work In Progress, draft-ietf-krb-wg-kerberos-referrals-00.txt
- [11] J. Linn, *Generic Security Service Application Program Interface Version 2*, Network Working Group (2000), RFC2743
- [12] Microsoft, *Step-by-step guide to Kerberos 5 (krb5 1.0) Interoperability*, <http://www.microsoft.com/windows2000/library/planning/security/kerbsteps.asp>
- [13] Brian Tung, Clifford Neuman, John Wray, Ari Medvinsky, Matthew Hur, and Jonathan Trostle, *Public Key Cryptography for Initial Authentication in Kerberos*, Work In Progress, draft-ietf-cat-kerberos-pk-init-06.txt
- [14] Samba Team, *Samba*, <http://www.samba.org/>

Predictable Management of System Resources for Linux

Mansoor Alicherry* K Gopinath†

*Department of Computer Science & Automation
Indian Institute of Science, Bangalore*

Abstract

In current operating systems, a process acts both as a protection domain and as a resource principal. This may not be the right model as a user may like to see a set of processes or a sub activity in a process as a resource principal. Another problem is that much of the processing may happen in the interrupt context, and they will not be accounted for properly. Resource Containers[1] have been introduced to solve such problems in the large-scale server systems context by separating out the protection domain from the resource principal by associating and charging all the processing to the correct container. This paper tries to investigate how this model fits into a Linux framework, especially, in the *soft real time* context. We show that this model allows us to allocate resources in a predictable manner and hence can be used for scheduling soft real-time tasks like multimedia. We also provide a framework in Linux which allows privileged users to have their own schedulers for scheduling a group of activities so that they can make use of the domain knowledge about the applications. We also extend this model to allow multiple scheduling classes.

1 Introduction

A general purpose operating system has to manage the underlying hardware resources (CPU, memory, disk etc) to provide a satisfactory performance for a mix of interactive, batch and, possibly, real time jobs. The system may be acting as a server providing specialized services to other computers. For many users perceived speed of computing is governed by server performance.

Modern high performance servers (eg. Squid) use a single process to perform different independent activities. However an activity may constitute more than one process. This may be for fault isolation and modularity. An example of this is CGI processing in http servers. In these cases,

the user may wish to control the scheduling and resource allocation for a group of processes together.

In current general purpose operating systems, scheduling and resource management do not extend to significant parts of the kernel[1]. An application has no control over the consumption of many resources that the kernel consumes on the behalf of the application. Whatever control it has is tied to assumption that each process is an independent activity.

In summary, in current general purpose operating systems, processes have the dual role of acting as a protection domain and as a resource principal. Hence, the notion of resource principal has to be separated from protection domain for better resource management. Resource containers[1], proposed by Banga, Druschel and Mogul, allow for fine-grained resource management by removing the role of resource principal from processes. Their work primarily concentrates on accounting correctly the overhead of network processing inside the kernel to the right resource container.

A resource container encompasses all system resources that a process or group of processes uses to perform an independent activity. All user and kernel processing for that activity is charged to the resource container and scheduled at the priority of the container.

In this paper, we study the management of system resources for Linux using resource containers. Though our design has many aspects similar to [1], our emphasis has been on the use of resource containers for scheduling soft real-time tasks¹ like multimedia. We extend the model of resource containers to support multiple scheduling classes. We also provide a framework in Linux for allowing different scheduling functions for different sets of applications. We also extend the APIs provided in the original model on resource containers. We also provide a */proc* interface to resource containers so that its parameters are easily accessible.

We have implemented *RCLinux*, a resource container

*The author is currently with Bell labs, Murray Hill, NJ. The author can be reached at mansoor@research.bell-labs.com

†The author can be reached at gopi@csa.iisc.ernet.in

¹Soft real-time tasks do not have "strict" deadline requirements, though some timeliness requirements still exist.

implementation for Linux version 2.2.5-15 (Red Hat 6.0). The patch and detailed documentation is available at <http://casl.csa.iisc.ernet.in/~mansoor/proj>.

1.1 Scheduling Anomalies in Current Operating Systems

Traditional schedulers have evolved along a path that has emphasized throughput and fairness. Their goal has been to effectively time-multiplex resources among conventional interactive and batch applications. But today, there is a growth of applications like multimedia audio and video, virtual reality, transaction processing etc. whose resource requirements are real-time in nature. Unlike conventional requests, real-time resource request need to be completed within application specific delay bounds, called deadlines, in order to be of maximum value to the application. For conventional and real-time tasks to co-exist, a scheduler has to allocate resources in such a way that real-time processes should be able to meet their deadlines, interactive jobs get good responsiveness and batch jobs should be able to make some progress. This is a very challenging problem.

Unix provides the “nice” system call to reduce/increase the base priority of processes so that there is forward progress in an application mix. But the values for these nice calls are often non-intuitive and many experiments have to be done to come up with the correct values. And these values are highly dependent on the application mix and have to be changed every time the application mix changes.

Another problem in the current general purpose operating system is that scheduling is done on a per process basis or on a per thread basis. A user having more number of processes/threads gets more CPU time than another user running a lesser number of threads. A malicious user can hog the CPU by creating lot of processes and hence preventing the progress of other users’ processes. We need a mechanism for preventing these types of “denials of service” by guaranteeing a fixed percentage of CPU for the users if required.

Yet another problem in the current general purpose operating systems is that network-intensive applications do most of the processing in interrupt context. Processing in interrupt context is either charged to the process that was running when the interrupt occurred or it is not charged at all. This can lead to inaccurate accounting and hence inaccurate scheduling.

Network servers have become one of the most important applications of large computer systems. Network processing in UNIX is mostly interrupt driven. Interrupt processing has strictly higher priority than user level code. This leads to interrupt live-lock or starvation when there is a high network activity.

Another problem in current general purpose operating system is in the lack of support for real time processes. Unix SVR4[2] supports static priority real-time classes, but

it has been shown to be of not much use[3]. Real time classes are supported by having a global priority range for each class, and real time class has higher priority value than the system class, which in turn have higher priority than the time shared class. Scheduling is done based on this priority. So when there is a process in the real time class, none of the system and time shared class processes are allowed to run. This model is fine for hard real time tasks, where the cost of missing a deadline is really high and efficiency is not of much concern. But this is not the right model for soft real time applications like multimedia, where the results are not catastrophic when the deadlines are not met. Also real time processes may depend on the system processes for some system services, but they are not able to get those services because of their own presence.

Nieh et al[3] studied the scheduling in SVR4 in the context of an application mix of typing, video and compute jobs. They found that no combination of assignment of different priority and classes to these applications gives a satisfactory performance to all the applications. They also found that the existence of a real time static priority process scheduler in no way allows a user to deal with problems of scheduling this application mix. They found that when using the real time class, not only do application latencies become much worse than desired, but pathologies can occur due to the scheduler such that the system no longer accepts user input.

For example, when the time shared class was used for all of the three type of application, compute bound (batch job) tasks performed well. This was due to the fact that the batch application forks many small programs to perform work, and then waits for them to finish. Since this job sleeps, the TS scheduler assumes it as an I/O intensive “interactive job” and provides it repeated priority boosts for sleeping.

1.2 What Useful Scheduling Models Exist?

Three useful concepts have been identified independently for their effectiveness in scheduling for applications in restricted domains[4]: best-effort real-time decision making, exploiting the ability of batch applications to tolerate latency and proportional sharing.

Best-effort decision making combines earliest-deadline scheduling with a unique priority for each request to provide optimal performance in under-load and graceful degradation in overload. Multilevel feed-back schedulers, as typified by UNIX time-sharing, take advantage of the ability of long running batch applications to tolerate longer and more varied service delays to deliver better response time to short interactive requests while attempting to ensure that batch applications make reasonable progress. Proportional sharing, also known as weighted fairness, has long been advocated as an effective basis for allocating resources among competing applications. Here each application is allocated resources proportional to its relative weighting.

1.3 Brief Overview of the Linux Scheduler

Linux supports 3 scheduling policies: SCHED_FIFO, SCHED_RR, and SCHED_OTHER. SCHED_OTHER is the default universal time-sharing scheduler policy used by most processes; SCHED_FIFO and SCHED_RR are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution.

A static priority value is assigned to each process and scheduling depends on this static priority. Processes scheduled with SCHED_OTHER have static priority 0; processes scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99.

All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned to its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

There is a single run-queue. The scheduler goes through each process in the queue and selects the task with the highest static priority. In case of SCHED_OTHER, each task may be assigned a priority or “niceness” which will determine how long a time-slice it gets. The “counter” attribute of each task determines how much time it has left in its time-slice. The scheduler selects the task with highest counter value as the next task to run. After every task on the run-queue has used up its time-slice (counter = 0), the counter for each task is set to the original priority + half the counter. In this way “interactive tasks” (tasks which were not in run-queue but whose counter was not zero) get a priority boost.

1.4 Related Work

Lazy Receive Processing (LRP)[5], proposed by Druschel and Banga, solves the problem of inaccurate accounting and interrupt live-lock due to network processing in interrupt context by identifying the process that caused the traffic and doing the network processing in the context of that process. In LRP, network processing is integrated into the system’s global resource management. Resources spent in processing the network traffic are associated with and charged to the process that caused the traffic. Incoming network traffic is processed at the scheduling priority of the process that receive the traffic and excess traffic is discarded early. Later on, the concept of resource containers[6], proposed by Banga, Druschel and Mogul, was introduced as a means of resource management in large-scale server systems. A prototype implementation was reported for Digital UNIX. A FreeBSD implementation of it is available in [7].

Goyal, Guo and Vin [8] propose an operating system framework that can be used to support a variety of hard and soft real-time applications in the system. This framework allows hierarchical partitioning of CPU bandwidth:

the OS partitions the CPU bandwidth among various application classes, and each application class, in turn, partitions its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. They used Start-time Fair Queuing (SFQ) algorithm for such a partitioning.

SMART[4], proposed by Nieh and Lam, a Scheduler for Multimedia And Real-Time applications, supports both real time and conventional computations and provides flexible and accurate control over sharing of processor time. SMART is able to satisfy real-time constraints in an efficient manner and provides proportional sharing across all real-time and conventional tasks. When not all real-time constraints are met, SMART satisfies each real time task’s proportional share, and adjusts its execution rate dynamically. SMART achieves this by combining the concepts of proportional sharing, latency tolerance, and best-effort real-time decision making. SMART uses an algorithm based on weighted fair queueing (WFQ) to implement weighted fairness. The concept of virtual time, biased with notion of latency tolerance, is used to measure the resource usage of the applications. The biased virtual time is then used as priority in the best-effort scheduling algorithm so as to satisfy as many real-time requirements as possible.

Resource Kernel[9], proposed by Rajkumar, Juvva, Molano and Oikawa, is another approach to real-time scheduling. A resource kernel is one which provides timely, guaranteed and enforced access to physical resources for applications. With resource kernel, an application can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is available to the application. A reservation can be time-multiplexed or dedicated. Time-multiplexed reservation is represented by parameters C, D & T, where T represents recurrence period, C represents processing time required within T, and D is the deadline within which the C units of processing time must be available within T. Oikawa et al[10] have an implementation of resource kernel for Linux.

RTLinux (RealTime Linux)[11] is an extension to Linux that handles time-critical tasks. In RTLinux, a small hard-realtime kernel and standard Linux share one or more processors, so that the system can be used for applications like data acquisition, control, and robotics while still serving as a standard Linux workstation.

KURT (KU Real-Time Linux)[12] is another approach to support real-time in Linux. KURT Linux allows for explicit scheduling of any real-time events rather than just processes. It has two modes of operation, the normal mode and the real-time mode. In normal mode, the system acts as a generic Linux system. When the kernel is running in real-time mode, it only executes real-time processes.

2 The RCLinux Resource Container Model

Resource containers[1] encapsulate all the resources consumed by an activity or a group of activities. Resources include CPU time, memory, network bandwidth, disk bandwidth etc. Our focus in this paper has been on CPU time.²

Scheduling and resource allocation is done via resource containers³. Processes are bound to resource containers to obtain resources. This *resource binding* between a process and a resource container is dynamic. All the resource consumption of a process is charged to the associated resource container. Multiple processes may simultaneously have their resource bindings set to a given container.

A task starts with a default resource container binding (inherited from its creator). The application can rebind the task to another container as the need arises. For example, a task time-multiplexed between several network connections can change its resource binding as it switches from handling one connection to another, to ensure correct accounting of the resource consumption.

Tasks identify resource containers through file descriptors. The semantics of these descriptors are the same as that of file descriptors: the child inherits the descriptors from the parent, and it can be passed between unrelated processes through the Unix-domain socket file descriptor passing mechanisms. APIs are provided for operations on resource containers. Security is enforced as an application can access only those resource containers that it can reference through its file descriptors.

We have added a new name space corresponding to the resource container: the *resource container id*, similar to the *pid* for processes, as it was not possible to access some of the resource containers through the existing APIs (e.g. a container that has no processes directly associated with it). This name space is available to users having the right privilege through the */proc* interface.

Resource containers form an hierarchy (figure 1). A resource container can have tasks or other resource containers (called *child containers*) in its *scheduler bindings* (i.e. the set of schedulable entities). The resource usage of a child container is constrained by the scheduling parameters of its parent container. On the top of the hierarchy is the root container. This encapsulates all the resources available in the system.

Hierarchical resource containers make it possible to control the resource consumption of an entire subsystem without constraining how the subsystem allocates and schedules resources among its various independent activities. This al-

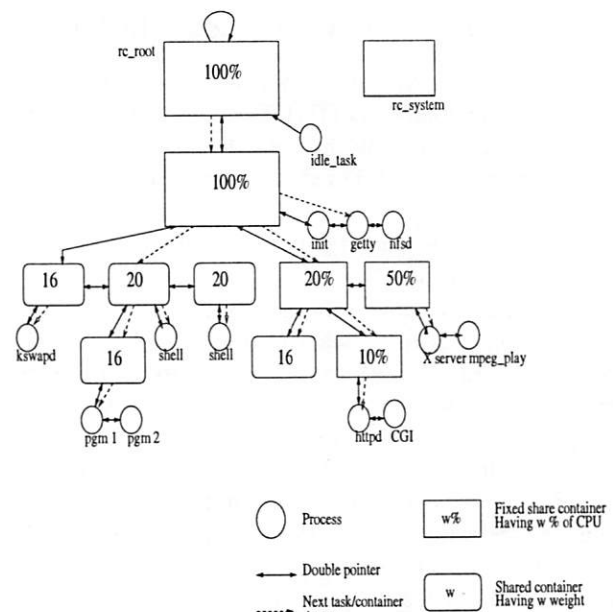


Figure 1: Resource container hierarchy

lows a rich set of scheduling policies to be implemented. Our Linux implementation allows the scheduling policies to be changed dynamically through APIs. It also allows these policies to be in dynamically loadable modules. This allows privileged users to try out various scheduling algorithms and to use better ones with a specific application mix.

The CPU resources allocated to a container may be either a fixed share of the resource its parent container is entitled to (called *fixed share child container*), or it may be shared with other children of the parent (called *shared child container*). In the case of shared children, the amount of CPU time it is entitled to is proportional to the weight of the container relative to other shared child containers of parent.

Along with fixed and shared child containers, our implementation support multiple scheduling classes. Scheduling classes have strict priorities, i.e. a container in the lower priority class will not be scheduled in the presence of container with a higher priority class.

For scheduling soft real-time processes, we need to attach them to fixed share containers. The CPU reservation of these containers have to be at least the amount of processing required for those processes. Unlike hard real-time scheduling, this will allow other time shared processes to have reasonable progress if the CPU reservation of soft-real time process is not 100%. Higher scheduling classes should be used only when absolutely necessary (eg. hard real-time) as this can possibly make all the processes in the lower class starve.

Our design does not take care of any interrupt live-locks that can occur due to high network activity. It has been shown that LRP[5] gives stable throughput under high load, hence we plan to incorporate it at a later time.

²We collect the resource usage of resource containers for CPU time, network bandwidth and disk usage. This can be used for computing the priority of the resource container if appropriate weights are assigned to each of the resource usage. Also the network bandwidth usage will reflect on the CPU usage since the protocol processing for the packets are significant and we account for the network processing properly.

³The descriptions in this section closely follow those in [1] & [6].

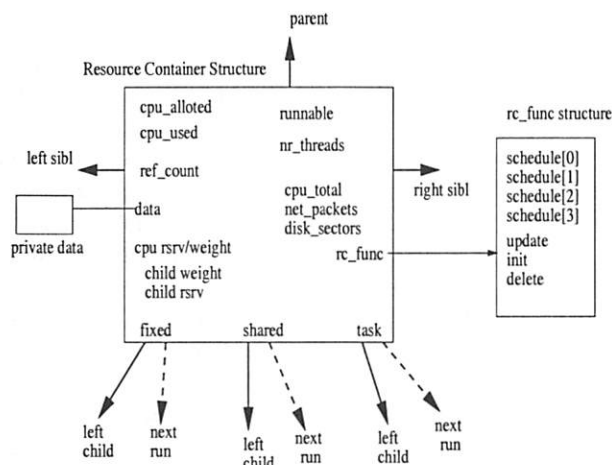


Figure 2: Resource container structure

2.1 Resource Container Data Structures

Resource containers are first-class kernel objects. They have various attributes that are required for scheduling and resource accounting along with functions for various operations on containers (figure 2). Some of the important attributes are:

1. *Mechanism Data*: This contains the necessary data of the container mechanism. This includes reference count, the list of runnable threads and next thread to run in the scheduler binding of the container, the list of child containers and next container to run in the fixed share children list and shared children list, and the parent container. The resource container hierarchy is implemented using a left-child, right/left sibling data structure, with each container having a pointer to its parent. The reference count keeps track of the number of references to the resource container. A reference to a resource container can be from a child container, from a task as its resource binding or from the global file table.

2. *Scheduler Data*: This contains data necessary for the scheduler. This includes the weight or CPU reservation that this container is entitled to from parent, sum of the CPU reservations and weights of the child containers, CPU time allocated to this container and CPU time used by this container if this container is in the *current path* (i.e. the path from root to the current container in the container hierarchy), number of runnable threads in the scheduler binding of the container, and a *runnable* bit mask. Each bit in the runnable bit mask indicates whether the container is *runnable* for the scheduling class corresponding to the bit. A container is said to be *runnable* for a class if there is a task in its scheduler bindings which belongs to that class or if it has a child container which is runnable for that class.

3. *Container specific functions*: Each resource container can have its own scheduler function for each of the scheduling classes and its own update, init and delete functions. The update function is used to update the scheduler data.

This is called by the scheduling routine of the child container before it gives the control of the CPU back to its parent (subsection 2.3). The init routine is called when a resource container is created or when the container specific function is changed by calling `set_rc_functions()` (section 2.5). This function may be used to initialise the resource container variables or to allocate space for container's private data. The delete function is called when the resource container is destroyed. This may be used to free the space allocated for private data. Each resource container has a pointer to a structure containing these function pointers.

4. *Statistics*: Statistics includes the total CPU usage of the container, the number of disk reads and writes, the number of network packets sent etc. Statistics will be propagated to the parent container, when the container is destroyed, if a flag `RCPROPSTAT` (i.e. propagate statistics flag) is set in the container.

5. *Private Data*: This can be used by the container specific scheduler and update routines.

2.2 Global Data Structures

To fully characterise a system-level resource container abstraction, we need additional global variables such as:

1. `rc_root`: This is the root container. All the resources in the system are allocated to the root container and hence to the hierarchy rooted here. Any container or task that has to get a resource has to be in the tree rooted here.

2. `rc_system`: This container is used to charge all the processing that could not be directly associated with any container (eg. network processing for a packet that is not destined to any of the processes). This container is not part of the hierarchy rooted at `rc_root`. The statistics information in this container can be used by the system administrators to monitor the system.

3. `rc_pool`: Pool of resource containers from where a resource container structure is allocated. When a resource container is freed, the resource container structure is returned to the pool.

4. `class_lastrun`: When a task of higher class becomes runnable, it preempts the currently running task. When the task of higher class is no longer runnable, then we have to restart the scheduling with the container where we had left off. This value is stored in `class_lastrun` array. If the element of `class_lastrun` that corresponds to the highest runnable class is `NULL`, the scheduling starts from root.

5. `rc_current`: This points to the currently chargeable container. This cannot be taken as the container to which the currently running task is bound, since we are providing a facility for the task to change its binding on the fly through a system call and during the call there is a change.

2.3 Resource Container Scheduler Framework

Each resource container has its own scheduling function (*rc_scheduler*) for each scheduling class. This can be in a loadable module and can be set on the fly using the *set_rc_functions()* system call (see section 2.5). The *rc_scheduler* function is passed the resource container and the class as the arguments. It returns the task that is to be scheduled next. The class is passed as an argument since it can use the same scheduling function for each of the classes. Similar is the reason for passing the resource container.

When *rc_scheduler* is called, the resource container structure will have two of its variables *cpu_allotted* and *cpu_used* set. This invocation of the scheduler is allowed to allocate (*cpu_allotted* - *cpu_used*) amount of CPU time to its child containers which are marked runnable for that particular class or to tasks in its scheduler binding belonging to that class. When it is allocating a CPU for a child container, it sets the *cpu_allotted* value of that container to the required amount and returns by calling the class specific scheduler routine of child container. If it is allocating CPU for a task, then it sets *cpu_allotted* in the task structure of that process and returns the process. When *cpu_used* is greater than *cpu_allotted* it first calls the update function of parent, with this container as the argument, and then returns by calling the class specific scheduler function of the parent. The update routine updates the *cpu_used* of the parent by adding to it *cpu_used* of the child and sets *cpu_allotted* and *cpu_used* of the child to 0. Any priority re-computation that has to be done is also done in the update routine.

2.4 rc_schedule() routine

This is the default scheduler function for all classes for all resource containers. This is written conforming to the scheduler framework given above. The scheduling starts with root.

The resource container allocates CPU for the containers having fixed CPU share, by an amount proportional to the CPU reservation of the children. Any remaining CPU time is allocated to the tasks in the scheduler bindings of the container. After that it will allocate the remaining CPU time for the time shared child containers, by an amount proportional to their weights. At anytime, if the CPU usage of the container exceeds the allocated time, or all the runnable containers and tasks are given the CPU then the control is passed to parent container by calling the class specific scheduling function of the parent.

The root has a different scheduler function; it returns NULL. This is because once all the child containers and tasks are scheduled by the resource container, it will call the schedule function of the parent to give back the control of the CPU. But for root, this cannot be the case as we have to restart the scheduling of the whole tree. One way

to achieve this is to check whether the container is root every time before calling the scheduler function of the parent container and restart the scheduling of the tree if it is root.

Another solution to this problem, which is cleaner and what we follow, is to have a different scheduler function for root. When the scheduler function for the container returns NULL, the *schedule()* function calls another routine *rc_root_schedule()* which restarts the scheduling of the tree, and returns the next thread to run.

rc_root_schedule() checks for the runnable flag of the root. If it is zero, then there are no processes in the system and it returns the *idle_task*. Otherwise it calls *rc_schedule()* with root and the highest runnable flag as the argument. If this function returns a task, that task is returned to *schedule()*. Otherwise it calls *rc_schedule()* again. This is because the root may be looking at some part of the tree (towards right) where there may not be any task in the highest class, so the first call to *rc_schedule()* will return NULL. The second call to *rc_schedule()* will restart the scheduling from the left most child and it will eventually find a task to run.

Similarly the update function for root is slightly different from update function of other resource containers, since the parent of root is root itself.

One important implementation issue that has a bearing on the design is that even if we are using regular tree traversal for scheduling, we cannot use any information on the stack between two context switches since the kernel stack changes for each context switch. So any information that is needed across context switches has to be kept in the resource container data structures itself. That is why *schedule()* calls *rc_schedule()* to find the next thread, rather than *rc_schedule()* itself acting as the sole scheduler function.

2.5 RCLinux Resource Container APIs

Resource containers can be accessed either through system calls or through the */proc* interface.

2.5.1 System Calls

System calls are provided for the users to make use of resource containers. Most of these system calls take the resource container descriptor as one of the arguments.

All the APIs except *set_rc_functions* were defined in [6]. But we have added the *pid* argument to *get_rc* and *set_res.binding* to make them more useful. *pid* was required as the argument to control the resources allocated to the processes which do not use the APIs (eg. an already existing application). This also gives the system administrator more control.

1. int rc_create(void)

This creates a new resource container and associates a file descriptor for the process with it. It returns the descriptor.

2. `int get_rc(int pid)`

This returns a descriptor for the resource container to which the process *pid* is bound. If *pid* is zero, the descriptor of the container of the calling process is returned. The permissions for obtaining the resource container descriptor of a process is the same as that of sending a signal to the process (ie. the calling process must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process). This will return a new descriptor every time this function is called.

3. `int set_rc_parent(int rcd1, int rcd2)`

Sets the parent of the resource container corresponding to *rcd1* to that of *rcd2*. It also checks whether the resources can be allocated to the child container from the new parent before changing the parent. It changes the scheduling parameters of the old parent, the new parent and the child container to reflect the change. The child container is removed from the binding list of the old parent and added to that of the new parent. This is used to add a newly created container to the resource container hierarchy, or to change the structure of the hierarchy.

4. `int set_res_binding(int rcd, int pid)`

Sets the resource binding of the process *pid* to the resource container corresponding to *rcd*. If *pid* is zero, the scheduler binding of the calling process is changed. Any future processing done by the process task will be charged to the new container. The task is removed from the scheduler bindings of the old container and inserted into that of the new container. Scheduling parameters of the containers are changed. The *need_resched* attribute of the task structure is set so that the task is preempted at the earliest safe point. The permissions to change the scheduler binding is the same as that of sending a signal.

5. `int set_fd_rc(int fd, int rcd)`

This binds the open file or socket corresponding to *fd* to the resource container corresponding to *rcd*. All the future processing for the file will be charged to the new container.

6. `int set_rc_opt(int rcd, struct rcopt *rco), int get_rc_opt(int rcd, struct rcopt *rco)`

Sets/gets the options of a resource container. Options includes scheduling parameters, statistics flags etc. *rcopt* is a structure that contains attribute type and attribute value as the fields.

7. `int set_rc_functions(int rcd, struct rcfunc *rcfp)`

Sets the class specific scheduler functions and update function of a resource container. *rcfunc* is a structure

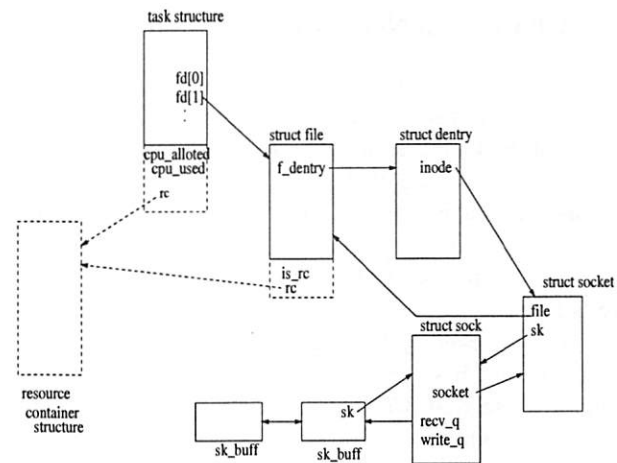


Figure 3: Kernel data structures (those dotted are new in RCLinux)

containing the function names. This allows a privileged user to load a module containing the functions and call this system call to use these functions.

3 RCLinux Kernel Modifications and Re-design

The RCLinux task structure *task_struct* contains a pointer to the resource container it is bound to (figure 3). Whatever resources are allocated to the resource container, it allocates for the processes in its scheduler binding and to its child containers.

Processes can access the resource containers through file descriptors (resource container descriptors). A pointer to the resource container and a flag was added to the *file* structure. If the flag is 1, then the file structure corresponds to a resource container. Otherwise, it has the normal Linux semantics (i.e. file, socket etc).

3.1 Changes to the File Management Subsystem

When a file is open, the resource container pointer of the file structure is set to the container to which current task is bound. The reference count of the resource container is incremented for each reference of the file structure (this is different from the reference count of the file descriptor since two file descriptors may point to the same file structure). Similar processing happens for the *socket()* call also.

In the close routine, the reference count of the associated resource container is decremented if there are no more references to the file structure.

3.2 Changes to Network Subsystem

Most of the protocol processing for incoming packets and for the outgoing packets during retransmission happens in the bottom-half handler *net_bh()*. In the current Linux kernel, proper accounting of this processing is not done. The network subsystem has been modified to do proper accounting. The Linux kernel already allows us to work out which socket file descriptor a network buffer (*skbuff*) structure belongs to, so we can always find the resource container for a packet by looking up the correct socket file. The accounting has been done by recording the timestamp counter (subsection 3.7) at the beginning and end of the bottom-half handler.

If the resource container corresponding to a network packet cannot be identified (eg. a packet to a port which no process is bound to), then it is charged to a separate container *rc_system* which is not part of the hierarchy rooted at *rc_root* (subsection 2.2). The number of bytes sent and received is also kept in the resource container as part of statistics.

3.3 Changes to *schedule()*

Whenever a process has to be preempted, Linux sets a flag in the task structure of the currently running process. When control reaches one of the safe points (eg. return from a system call), the *schedule()* function is called to determine the next process to be scheduled and to context switch to that process. The *schedule()* function has been changed to support the resource container framework.

The *schedule()* routine updates the CPU usage of the current resource container by adding to it the CPU usage of the currently running process. Then it checks whether any process has become runnable in any higher class than that of the current container. If there is any, the highest class runnable resource container is the next resource container to be scheduled. Otherwise the current container is scheduled.

Once the next resource container is found, the class specific scheduler function of that container is called to decide on the next task to be run. This function need not necessarily return a task in its own scheduler binding since it could allocate the CPU to a child container and call its scheduler function, or its CPU time allotted could have expired by this time, in which case it calls the scheduler function of the parent to give back the control of the CPU (subsection 2.3).

If the class specific scheduler function of the next resource container to run returns NULL, then scheduling restarts from the root container.

After finding out which task to schedule next, the container to which this task is bound is saved. If the next task to be scheduled is not the currently running one, a context switch is effected.

3.4 Changes to *fork()* and *exit()*

Linux supports a *clone* system call that an user can use to create a clone of a process. A flag is passed as an argument to the *clone* call that specifies the type of the objects (eg. virtual memory, signal mask etc) that are to be shared. Both *fork* and *clone* system call make use of the same function but use different flags. A new flag *CLONE_RC* has been added to the clone flags. If *CLONE_RC* is set then this function allocates a new resource container for the child process and makes it the child of the resource container of the parent process. Otherwise the resource container of child process is set to that of the parent process. When the child process is made runnable, it is added to the scheduler bindings of the resource container to which it is bound to.

When a process exits, Linux frees most of the resources (eg. virtual memory, file tables etc). This cannot free the complete task structure since some of the information is required later (eg. exit code for the parent, fields in task structure used by the scheduler, and the kernel stack since the process is still the “currently running” process). These fields are freed when the parent calls *wait*. A resource container which is bound to the exiting task cannot be freed during exit as it is needed by the scheduler. It is freed when parent calls the *wait*.

3.5 System Initialization

During system initialization the resource container initialization routine *rc_init()* is called. This initializes the root container. It allocates *RCMAXCPU* to the root container, which is the value that is used to schedule all the processes system wide in one traversal of the hierarchy. It sets the cpu reservation of the container as the maximum reservation possible. The resource container for “idle task” is set to root, but the task is not added to the scheduler bindings of the root since we do not want it to be scheduled when there is some other task in the system. The parent container of the root is set to root itself. Current running container is set to the root container.

Process 0 creates *init* process with the *CLONE_RC* flag set so that a new container is created for *init* task. The scheduling parameters of this new container is set so that it is entitled to all the resources root can have. *Init* process spawns other kernel threads with *CLONE_RC* flag set.

3.6 */proc* Interface

Linux provides, in a portable way, information on the current status of the kernel and running processes through the */proc* interface. We have changed */proc* to support resource containers.

Resource containers are represented as directories in */proc*, named *rc<resource_container_id>*. The files in this directory are a read only file *status* and a write only file *cmd*.

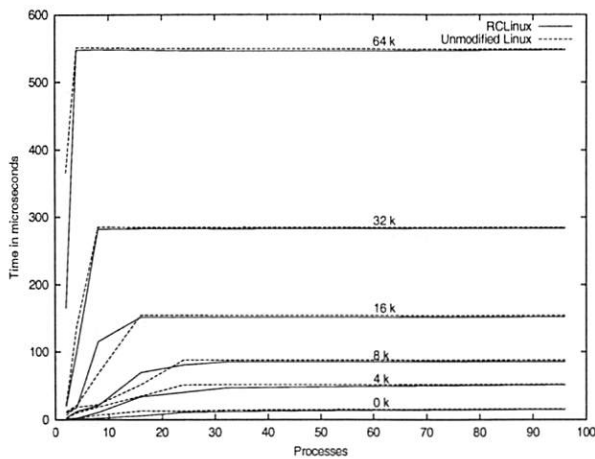


Figure 4: Context switch time for RCLinux and Unmodified Linux for processes of various sizes

We can get various attributes of a resource container like the list of child containers, the list of child processes, resource usages etc by reading the corresponding *status* file. This information can be used for debugging and system administrative purposes.

The file *cmd* is used for giving various commands to the resource container. Here the name of the API and its argument is directly written to the file. This helps in easy use of resource container functions.

3.7 Other Changes

One of the important requirements for the success of proper accounting and scheduling is the ability to keep track of time accurately. Linux uses a clock which ticks HZ times per second with HZ set as 100. This will not help us to do proper accounting of the activities since we get only a resolution of 10ms. Modern CPUs provide a timestamp counter which is incremented for every CPU clock. We have used the timestamp counter for accurately measuring the time.

Another change we have made to the Linux kernel is to the routines which insert or delete a process on the run queue. A process is inserted/deleted to/from the run-queue of the resource container to which it is bound, rather than to a global run-queue. Also whenever a process is inserted to (or deleted from) a run-queue, the *runnable* flag of the associated resource container is updated and this updating is propagated till the root.

Whenever a resource container is freed, the *last_run* array is examined and all the elements of the array pointing to the container is set to NULL.

operation	RCLinux	Linux
Simple syscall	0.71	0.71
Simple read/write	1.04	1.05
Simple stat	5.94	5.88
Simple open/close	7.66	7.54
Select on 100 tcp fd's	35.95	35.32
Signal handler installation	2.18	2.23
Signal handler overhead	2.92	2.92
Protection fault	1.48	1.48
Pipe latency	7.95	9.30
Process fork+exit	394.57	440.46
Process fork+execve	4109.00	4139.00
Process fork+/bin/sh	15023.00	14977.00

Table 1: Time taken for various operations(in micro seconds)

4 Experimental Evaluation

4.1 Performance overhead

We ran *lmbench*[13] to evaluate the performance of RCLinux compared to an unmodified version of Linux. The tests were performed on a 400 MHz Celeron with 128MB memory. We found that the overhead was very minimal. The results are summarized in table 1.

Figure 4 shows the context switch time for different number of processes for different sizes. The processes are connected in a ring of Unix pipes. Each process reads a token from its pipe, possibly does some work, and then writes the token to the next process. A size of zero is the baseline process that does nothing except pass the token on to the next process. A process size of greater than zero means that the process does some work before passing on the token. The work is simulated as the summing up of an array of the specified size. The graph shows that RCLinux performs slightly better when number of processes is large.

4.2 Use of fixed CPU share

In this experiment, we create three child containers C1, C2 and C3 for the resource container to which *init* is bound, having a fixed CPU share of 16%, 32% and 48% respectively. We also create three CPU bound jobs R1, R2 and R3. The cumulative execution time of these jobs are shown in figure 5.

The figure shows that when all the processes are running, R1, R2 and R3 get 14.3%, 28.6% and 42.9% of the CPU respectively. There is a proportional decrease in the CPU allotted than what has been specified and may be due to the lower resolution of the CPU timer⁴, pre-emption being done only at safe points and the non-accounting of some of the activities in the system (eg. network activities that are

⁴Eventhough we use CPU time stamp counter for keeping track of time, we get timer interrupts only in 10ms intervals.

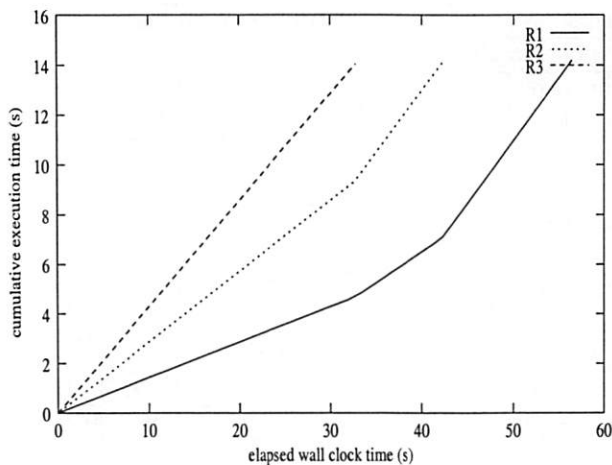


Figure 5: Proportional allocation of CPU for fixed share containers

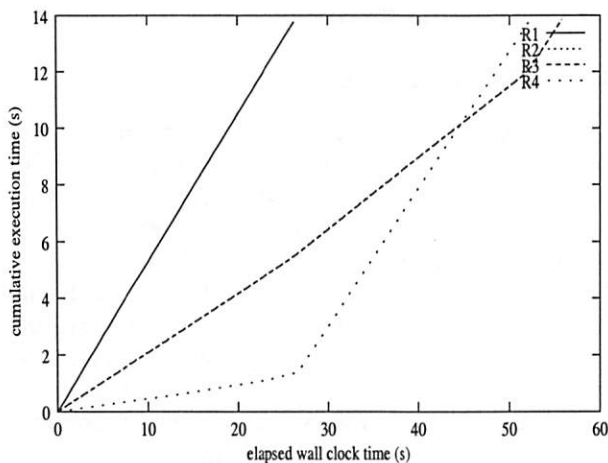


Figure 6: Proportional allocation of CPU for fixed share and shared containers

charged to *rc_system*, the time taken for scheduling etc.). Once R3 exits, R1 and R2 get 25.2% and 50.1% of the CPU respectively and the remaining CPU power is used by various system activities and other processes in the system. This is more than what has been specified as there are not many other activities in the system. Once R2 also exits, R1 gets 50.0% of the CPU.

4.3 Allocation of CPU in presence of fixed share and shared containers

Another experiment was conducted in which three fixed share containers and a shared container were created. Fixed containers had a CPU share of 48%, 16% and 16%. CPU bound jobs R1, R2 and R3 were bound to these containers. Another CPU bound job R4 was bound to a shared container. All the four jobs were same. The cumulative execution time of these containers are shown in figure 6.

When all the processes were running, R1, R2, R3 and

R4 got 54.2%, 20.9%, 20.9% and 8.7% of CPU time respectively. When R1 exited, the shared container got most of the CPU that R1 was using. R2 and R3 got 25.2% and R4 got 48.9% of the CPU. R4 completed before R2 and R3. When R4 exited, R2 and R3 got 49.5% of the CPU.

4.4 Scheduling for Multimedia

Multimedia applications often perform poorly under Linux when there are some CPU bound processes running. For this experiment, we have run a multimedia application in the presence of CPU bound processes - each being a distributed genetic algorithm client⁵. *mpeg_play* was used for viewing the mpeg file. Frame rate was noted for different number of genetic algorithm clients. The frame rate observed under Linux was 15.5, 9.8, 7.0 and 5.6 for 1, 3, 5 and 7 clients respectively.

Even if the frame rate was 15.5 with one client, the movie was jerky. This was because of the way Linux schedules processes. Initially both *mpeg_play* and the client will be allocated 200ms (ie *count* = 20) CPU time. Once *mpeg_play* is scheduled, it will require service from the X server and it will sleep. Next the client is scheduled, and it will use up its 200ms since it does not sleep in between. Again, *mpeg_play* is scheduled and it will again sleep for service from X server. So the X server and *mpeg_play* will be scheduled alternately till their remaining CPU quanta is exhausted (ie *count* becomes 0). After this, the whole cycle repeats. The movie is jerky as the client runs once per this major cycle.

Next, we ran *mpeg_play* by creating two resource containers with fixed cpu share and binding the *mpeg_play* and X server processes to those containers. We tried various amounts of CPU shares from 10% to 60%. We also allocated 30% to 50% of the CPU to X server. But the maximum frame rate went only up to 7.0.

The problem in this case was that even if the scheduler was allocating enough CPU time to both *mpeg_play* and X server, they were not able to use it since one needed the service of the other. After *mpeg_play* generated a frame, it required the service of the X server to display it. After displaying the frame, the X server needed *mpeg_play* to be scheduled to generate the next frame.

Now we ran *mpeg_play* by creating a single resource container with fixed cpu share and binding both *mpeg_play* and X server processes to that container. We tested it by allocating different amount of CPUs to this resource container and running distributed genetic algorithm with different number of clients.

With this arrangement for binding processes, we obtained significant performance for CPU shares more than 60%. Also the picture was very smooth (no jerks) even for

⁵This application recognizes digits using a number of computational nodes (clients) with the server using many clients. Most of the time server sleeps. We have run all the clients on the same machine.

No. of clients	CPU share								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
1	4.7	5.2	10.0	10.4	15.1	15.5	19.9	20.7	25.3
3	4.7	5.2	9.9	10.4	15.0	15.6	19.9	20.7	25.3
5	4.7	5.2	10.0	10.3	15.0	15.6	19.9	20.7	25.3

Table 2: Frame rate using resource containers

lower frame rates. The resulting framerates are shown in Table 1. The output shows that the frame rate is independent of number of clients running in the system and depends only on the CPU allocated to the resource container.

5 Conclusions and Future Work

The concept of resource container was introduced to address the lack of appropriate support for server applications in existing operating systems. We have extended it by introducing multiple scheduling classes and by using container specific schedulers. We have shown that we can use resource container mechanism for predictable performance of applications and for scheduling multimedia applications.

We have implemented resource containers for Linux by making minimal changes to the existing kernel which is designed under the assumption that a process is a resource principal. We have modified only two kernel data structures, *task_struct* and *file*. We have changed a total of 12 .c files, 3 .h files and 1 .S file in the kernel, fs, net, init and arch directories.

The statistics information that is maintained for disk accesses and network packets can be used for taking better CPU and I/O scheduling decisions. A detailed study has to be conducted to find out the influence these have on the priority computations.

We have not looked into memory management subsystem in this resource container framework. When a group of processes are performing related activities, the working set of each process may depend on the working set of the other processes in the group. So better paging decisions may be possible if resource container framework is added to the memory management subsystem. We have also not looked into multiprocessor related issues in our design.

Acknowledgements: We thank N.Ganesh for his help in running lmbench on RCLinux and Linux-2.2.5 and Stephen Tweedie, our shepherd, for his many suggestions on improving the paper. Financial support from Veritas Software, Pune is also gratefully acknowledged.

References

- [1] Gaurav Banga, Peter Druschel, and Jeffery C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of Third*

Symp. on OS Design and Implementation, New Orleans, Feb 1999.

- [2] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall of Australia Pty Ltd, 1994.
- [3] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Proc. of Fourth International Workshop on Network and OS Support for Digital Audio and Video*, Nov 1993.
- [4] Jason Nieh and Monica S. Lam. The design of SMART: A scheduler for multimedia applications. CSL-TR-96-697, Stanford University, Jun 1996.
- [5] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. of Second Symp. on OS Design and Implementation*, Seattle, WA, Oct 1996.
- [6] Gaurav Banga. *Operating System Support for Server Applications*. PhD thesis, Rice University, May 1999.
- [7] Resource containers and LRP for FreeBSD. <http://www.cs.rice.edu/CS/Systems/ScalaServer/code/rescon-lrp/README.html>.
- [8] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proc. of Second Symp. on OS Design and Implementation*, Seattle, WA, Oct 1996.
- [9] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa. Resource kernels: A resource-centric approach to RT systems. In *Proc. of SPIE/ACM Conf. on Multimedia Computing & Networking*, Jan 1998.
- [10] Shui Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symp. Work-In-Progress*, Dec 1998.
- [11] RTLinux. <http://luz.cs.nmt.edu/~rtlinux>.
- [12] KURT: The KU Real-Time Linux. <http://hegel.ittc.ukans.edu/projects/kurt>.
- [13] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proc. of 1996 USENIX Technical Conf.*, San Diego, CA, Jan 1996.

Scalable Linux Scheduling

Stephen Molloy, CITI - University of Michigan

`<smolloy@engin.umich.edu>`

Peter Honeyman, CITI - University of Michigan

`<honey@citi.umich.edu>`

Abstract

For most of its existence, Linux has been used primarily as a personal desktop operating system. Yet, in recent times, its use as a cost-efficient alternative to commercial operating systems for network servers, distributed workstations and other large-scale systems has been increasing. Despite its remarkable rise in popularity, Linux exhibits many undesirable performance traits.

Concerned about the scalability of multithreaded network servers powered by Linux, we investigate improvements to the Linux scheduler. We focus on pre-calculating base priorities and sorting the run queue for efficient task selection. We propose an improved scheduler design and compare our implementation in terms of scalability and performance to the existing Linux scheduler. Our analysis shows that improvements can be made to the existing scheduler without introducing overhead, thus improving the scalability and robustness of the Linux operating system.

1. Introduction

Linux, a strong and steadily increasing presence on the Internet today [4], commonly provides cost-effective and load-tolerant solutions for network services. Its familiarity to UNIX users, source code availability, and ability to run on many different architectures explain Linux's rapid increase in popularity. Many software companies, including AOL, Netscape and IBM [6, 8], offer Linux products. Several organizations use Linux on routers, print and file servers, firewalls and, of course, web application servers [10].

At the same time as Linux's popularity has increased, the use of Java for web applications has grown immensely. Java technology is a key component in building scalable application servers. However, communications-intensive Java applications often create large numbers of threads and Linux does not handle such stress gracefully.

Concerned about the scalability of multithreaded network servers powered by Linux, we investigate improvements to the Linux scheduler. Experiments by

IBM indicate that as much as 30% of the total CPU time in the system is spent in the scheduler when the number of running threads is high [2]. Our analysis shows the current scheduler uses an expensive and redundant algorithm for task selection. Our goal is to improve the scalability of the Linux scheduler to adapt it to enterprise-scale server workloads. Our analysis shows that our new scheduler implementation achieves these goals.

The rest of this paper is organized as follows. Section 2 gives some background information on our project and provides some insight as to why we chose to tackle the scheduler rather than the Linux threading model. Section 3 describes Linux's current approach to scheduling. Section 4 explains the problems with it. Section 5 outlines our approach to solving the problem and Section 6 describes its performance relative to the current Linux scheduler. All kernel modifications, experiments, and descriptions are against a 2.3.99-pre4 Linux kernel and, when used, Java version 1.1.7 of IBM's JDK.

2. Background

Because Linux grew from a desktop operating system, many issues prevent it from being a dominant force in the enterprise server market. The design and implementation of Linux has traditionally focused on simplicity and versatility rather than small performance gains and scalability. The implementation of the Linux thread model and scheduler illustrate this approach.

The Linux thread model is a one-to-one model, meaning that every user-level thread is mapped onto its own kernel thread. While this model makes programming in the kernel less complicated, it sometimes forces programs to generate more kernel threads than is necessary. Forcing the kernel's default scheduler to accommodate too many threads can adversely affect a server's performance.

Many established operating systems support many-to-one or many-to-many thread models in which each kernel thread has many user level threads mapped to it. In these models, a secondary scheduler chooses which of the mapped user level threads to run. The multi-tier scheduling approach of these models assures that the scheduler at each level will be faced with a more manageable number of threads.

The Linux scheduler, like its thread model, is also an exercise in simplicity. The heart of the scheduler is concisely coded in just a few lines that evaluate runnable threads and then picks the best. The price for this simplicity though, is a linear time algorithm that repeats many of the same calculations that were performed on its last invocation.

In this project, we address the shortcomings of the scheduler rather than those of the threading model. The reasons for this decision are simple. We saw the redundant calculation and $O(n)$ loop in the current scheduler and knew we could improve it. We also know that the Linux kernel community has been very protective of its threading model in the past and we wanted to avoid upsetting any contributors. Finally, the original timeline for the project called for a working design and implementation within the time frame of one semester. Developing a new threading model for Linux would almost certainly require more time than we had available.

Other groups have spent considerable time designing alternative schedulers for Linux [1, 5, 9]. Linux discussion groups provide evidence that the scheduler has been and continues to be an interesting topic for the developer community. However, most alternative scheduler designs focus on reducing latency for real-time processes rather than improving the overall scalability of the default scheduler.

volatile long	state
unsigned long	policy
long	counter
long	priority
struct mm_struct	*mm
struct list_head	run_list
int	has_cpu
int	processor

Table 1: This table shows the fields of the task structure that are most relevant to Linux scheduling.

While it is our goal to improve scalability and performance of the scheduler when faced with a large number of runnable threads, it is not our intent to change the criteria it uses for thread selection. We feel that the current criteria are carefully chosen and sufficient to make good decisions with a minimum amount of calculation. Our primary concern is simply that these criteria are not being used in an optimal algorithm by the scheduler.

In the remainder of this paper, because Linux uses a one-to-one threading model, we do not distinguish between a user thread and a kernel thread. Also, to match the terminology used in the kernel source code, we refer to any thread in the system as a task.

3. Current Scheduler

To understand why the current Linux scheduler scales poorly with the number of runnable threads in the system, it is necessary to be familiar with its data structures, algorithms, and conventions. This section outlines the existing scheduler to clarify our observations and design decisions.

3.1 Task Structure

The basic execution context in Linux is referred to as a task. The task structure is responsible for maintaining a task's address space information, whether that address space is shared with other tasks, and other state information about the task and its registers. It also tracks task statistics for memory management and resource control, privileges, file descriptors, signal handlers and other task specific information. The various fields of the task structure used in the scheduler are illustrated in Table 1.

The task's `state` field can be set to one of six values, each representing a different state that in which a task might find itself (such as blocking or sleeping.) `TASK_RUNNING` is the value of `state` when a task is runnable.

The `policy` field is set either to `SCHED_FIFO`, `SCHED_RR` (round robin) or `SCHED_OTHER` to determine the scheduling policy for the task. The first two options are for real-time tasks, while the third is for all other tasks. Real time tasks are always run before regular tasks if they are runnable. The `policy` field is also used to track yielded tasks. When a non-real-time task gives up its processor via the `sys_sched_yield()` system call, a bit (`SCHED_YIELD`) in the task's `policy` field is set so this information can be passed on to the scheduler.

The field `has_cpu` is set to 1 while a task is executing on a processor and 0 otherwise. Upon setting `has_cpu`, the field `processor` is set to the processor ID on which the task will execute. The task structure also contains pointers that identify the memory map in which it runs and its place on the run queue.

The two most important factors in determining which task executes next are represented by the `priority` and `counter` fields. `Priority` is an integer between 1 and 40. Higher numbers represent higher priority. Twenty is the default value for all tasks. (Real-time tasks also use a priority value, but it ranges from 0 to 99 and is stored in a separate field called `rt_priority`.) `Counter` is a value that indicates the time remaining in the task's current quantum. `Counter`, measured in 10ms ticks, can range from zero to twice the task's priority. Linux uses this field to enforce a fairness policy.

It is worth noting that all tasks, whether they are lightweight threads or full-fledged processes, are treated the same by a Linux system. All processes and threads are visible in various system status commands such as `ps` and `top`. Consequently, the default scheduler, which is responsible for accommodating all tasks in the system, can be placed under considerable stress when running multithreaded applications.

3.2 Run Queue

The run queue in Linux is a circular, doubly linked list containing all tasks in the `TASK_RUNNING` state. The scheduler traverses this list when it looks for a task to run. The list is not maintained in sorted order. When the scheduler finds two equivalent tasks, the one closer to the front of the list is chosen. Newly created or awakened tasks are placed at the beginning of the run queue. The list is doubly linked and circular, so tasks can also be added to the end of the run queue.

3.3 Schedule()

The Linux kernel function `schedule()`, as in other operating systems, is called from over 500 places within the kernel, underscoring its significance to

overall system performance. The `schedule()` function is called by a task when it yields the processor, blocks for I/O, expires its quantum, or is preempted by another (higher priority) task. `Schedule()` uses the execution context of the task that called it (referred to as the previous task in the scheduler). `Schedule()` is charged with finding the best task to take the previous task's place on the processor. In doing so, it makes use of a heuristic computed by the function `goodness()`.

3.3.1 Goodness Calculation

The scheduler uses the `goodness()` function to determine the utility of running a given task. A high goodness value means it would be a sound decision to run the given task next. For tasks that are marked `SCHED_FIFO` or `SCHED_RR`, `goodness()` returns 1000 plus the value stored in the task's `rt_priority` field. For other tasks, however, `goodness()` returns a much lower number and shows more discretion in its evaluation.

For `SCHED_OTHER` tasks, four factors are taken into consideration. The first factor is a task's counter value. If a task has a counter value of zero, then `goodness()` returns a utility of zero. This lets the scheduler know a runnable task was found but its time slice is used up. If a task's counter value is not zero, then its goodness value is set to the sum of its counter and priority values.

The third and fourth factors are bonuses for processor affinity and sharing an address space with the previous task. A small, one point advantage is given to tasks that share memory maps, because of the reduced overhead for the context switch. A somewhat larger (15 point) bonus is given to tasks whose last run was on the current processor, to try to take advantage of memory lines that may still reside in the processor's cache. These bonuses are added to the previously calculated goodness value to determine the task's final goodness value.

3.3.2 Scheduling Algorithm

The scheduler begins by executing all outstanding bottom-halves (delayed functions that were too substantial to run during an interrupt.) After some additional administrative work, the scheduler enters the heart of its code: an examination of all runnable tasks. The previous task is the first task looked at by the scheduler. If the `SCHED_YIELD` bit is set for the previous task, then the scheduler clears the bit and uses zero as the task's goodness value. Otherwise, it calls `goodness()` to determine this value.

Next, the scheduler walks through the run queue, evaluating the goodness of each task not currently

running on another processor. After all runnable tasks have been examined, the task with the greatest goodness value is chosen to run on the processor. If no task has a goodness greater than zero¹, then the scheduler jumps to a piece of code responsible for recalculating the counter values of all tasks in the system (runnable or otherwise) and returns to search the run queue again.

While the `goodness()` function by itself is very simple, executes quickly and considers the most appropriate factors in making intelligent scheduling decisions, it is expensive to recalculate `goodness()` for every task on every invocation of the scheduler.

4. Problem

Efficient handling of multiple threads is crucial for enterprise servers to make best use of system resources, communicate with many parties at the same time, and reduce the average time that service requests spend waiting for an available server. Multiplexing I/O system calls (such as `select`) can help in some situations, but they are not always available. The popular Java programming language is a prime example.

Threads are an essential element in the Java language: because the Java language lacks an interface for non-blocking and multiplexing I/O, threads are especially important in constructing communications intensive applications. Typically, one or more Java threads are constructed for each communications stream used by a Java program. Therefore, a natively threaded Java Virtual Machine (such as IBM's JVM [7]) can put a strain on the Linux scheduler, which, as we have seen, examines the goodness function for every thread in the run queue. This can be an exhausting process.

Experiments at IBM show the impact of the Linux scheduler on the performance of a multithreaded network application written in Java [2]. VolanoMark is a benchmark written to measure the performance of VolanoChat, a Java implementation of a chat room server. Because its results have been widely published in magazines such as JavaWorld [3], VolanoMark is an important benchmark for comparing the performance of different implementations of the Java Virtual Machine.

The VolanoMark benchmark establishes a socket connection to a chat server for each simulated chat room user. Because Java does not provide non-blocking read and write, VolanoMark uses a pair of

threads on each end of each socket connection (4 threads per connection) to simulate non-blocking I/O. For a 5 to 25-room simulation, the kernel must potentially deal with 400 to 2,000 threads in the run queue. The key measure of performance reported by VolanoMark is message throughput, i.e., the number of messages per second (over all connections) the server is able to handle during a benchmark run. The measurements for IBM's report were taken while running VolanoMark over a loopback interface, eliminating any network overhead involved; the heap size for the test was large enough for the overhead of Java garbage collection to be less than 5% of the total elapsed time throughout the experiments.

The results of the VolanoMark experiments show that 25-room throughput decreased by 24% from 5-room throughput due to the additional threads in the system. A profile of the kernel taken during the VolanoMark runs showed that between 37 (5-room) and 55 (25-room) percent of total time spent in the kernel during the test is spent in the scheduler.

5. ELSC Scheduler

To reduce the amount of time spent in the scheduler we developed a new scheduling solution, called the ELSC scheduler. Our goals in implementing this scheduler are as follows:

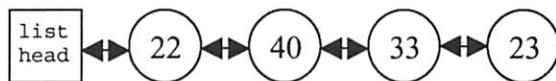
- 1) Keep changes local to the scheduler. Do not change current interfaces to the scheduler.
- 2) Keep the concept and implementation simple.
- 3) Behave like the current scheduler as much as possible.²
- 4) Maintain existing performance for light loads. Scale gracefully under heavy loads.

The ELSC scheduler is a table-based scheduler that keeps the run queue in a sorted order, making scheduling decisions easier and faster. We chose a table based design because it relieves us of the overhead of sorting lists. It also avoids complexity when inserting or removing tasks, unlike, say, a heap.

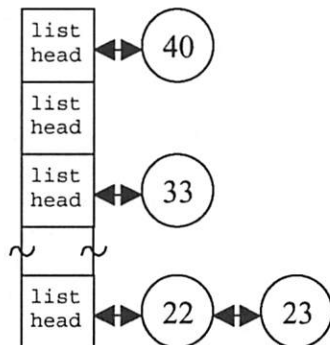
The foundation of the ELSC scheduler is its ability to keep tasks in an order that makes choosing one fast. The key to this sorted order is in how a task's `goodness()` value is calculated in the current scheduler. The `goodness()` calculation consists of a static and a dynamic part. The static part consists of a task's priority and counter values. While a task is

¹ The run queue must contain at least one task for this condition to count. An empty run queue will schedule the idle task rather than trigger the recalculation.

² By behave, we mean that if the current scheduler always selects a real-time task over a `SCHED_OTHER` task, even if it has a zero counter, then the ELSC scheduler should do the same. Aside from a few optimizations, the ELSC scheduler does adhere to the same quirky rules as the current Linux scheduler.



(a) Run Queue for Current Linux Scheduler



(b) Run Queue for ELSC Scheduler

Figure 1: Illustration of run queue structures for both schedulers. The squares represent list heads and the circles represent tasks. The labels on the tasks indicate the static goodness of that particular task.

on the run queue but not running on a processor, its counter value does not change. Likewise, its priority almost never changes, though when it does, the ELSC scheduler adapts accordingly. We refer to the combination of these two values as a task's *static goodness*. A task's *dynamic goodness* consists of memory map and processor affinity. Despite the fact that they don't change while a task is on the run queue, they depend on which task and processor are calling `schedule()`. The ELSC scheduler uses static goodness to sort tasks on the run queue.

5.1 Implementation

The ELSC scheduler uses a new structure for the run queue. Previously, the run queue was a simple doubly linked list of nodes that each point to a task as shown in Figure 1a. To make scheduling decisions fast, we need to keep the run queue sorted, while at the same time keeping insertion and deletion times small. The ELSC scheduler does this with an array of 30 doubly linked lists. Each list in the array is used to hold tasks in a certain static goodness range, as demonstrated in Figure 1b. Lists at one end of the table hold tasks with the highest static goodness values while the other end hold tasks with the lowest. A `top` pointer is used to indicate the highest priority list that contains a runnable task.

To change the structure of the run queue from a single list to a table of lists, we need to change four run queue manipulation functions as well: `add_to_runqueue()`, `del_from_runqueue()`, `move_first_runqueue()` and `move_last_runqueue()`. The first of the two

functions puts tasks on and removes them from the run queue when appropriate. The next two tasks give a task an advantage/disadvantage in the selection process when another task has the same `goodness()` value. Only `schedule()` manipulates the run queue directly.

The function `add_to_runqueue()` is modified slightly to deal with the new table structure. Like the current scheduler, it adds tasks to the front of a list. The particular list depends on the task. If the task is real-time, it uses one of the ten highest lists, determined by dividing the `rt_priority` field by 10. If the task is a `SCHED_OTHER` task, then the list is determined by adding `counter` to `priority` and dividing by four. Once the list is chosen, the task is added to the front of that list and the `top` pointer is updated if necessary.

When all tasks in the run queue exhaust their time quantum, their counters are all zero. At this time, the current scheduler resets all counters in the system. The ELSC scheduler does the same. However, to avoid re-indexing every task in the run queue when their counter is reset, we modified `add_to_runqueue()` as follows. If the task being inserted has a non-zero counter value, the task is inserted as described above. Otherwise, `add_to_runqueue()` uses a predicted counter value for the task, based on its knowledge of how the scheduler resets them. Using the predicted counter value and its current priority, the task is indexed into the run queue and added to the end of its list. This way, all zero counter tasks reside at the end of the list, behind all tasks with a non-zero counter value. The zero counter tasks are out of the way of the scheduler, but are in position once all other tasks in the run queue exhaust their quanta. A `next_top` pointer is used to keep track of the highest priority list containing a runnable task after counters are reset and is set at this time.

In the current scheduler, the `del_from_runqueue()` function removes a task from the list it is on by simply pointing the two nodes on either side of it in the list at each other. Then it sets its own run queue node's `next` pointer to `NULL`, indicating that it is no longer on the run queue. The ELSC scheduler follows exactly the same process. Afterwards, it updates both the `top` and `next_top` pointer if the removal of the task caused either one of them to change. In the ELSC scheduler, it is possible for a task to be considered on the run queue but not actually be in one of the lists in the table.³ Because a node's `next` pointer indicates presence on the run queue by the current scheduler, the ELSC

³ The reason for this is because we actually remove tasks from the run queue while they are running, but the rest of the Linux system would like to think that they are still on the run queue. This gives us a way to tell precisely if a task is on a list.

scheduler also sets the `prev` pointer to `NULL` to indicate that the task is not actually on any list, thus leaving the `next` pointer alone if the task is considered “on the run queue” without being on the run queue.

The functions `move_first_runqueue()` and `move_last_runqueue()` were meant to bias decisions in the case of a `goodness()` tie. Consequently, we need only to move tasks within their current lists in the table. A task is moved within its current list to the beginning or end of its section of the list. Recall that lists can contain tasks with both zero and non-zero counter values. These functions behave appropriately when faced with mixed-counter lists.

In addition to the modification of these four functions, code was added to initialize the run queue table structure when booting. We also wrote two test routines that determine whether a list contains tasks with zero or non-zero counter values.

5.2 ELSC Scheduling Algorithm

Like the current scheduler, the actual ELSC implementation of `schedule()` begins by executing all outstanding bottom-halves and then performing some additional administrative work. It then deviates from the current scheduler as follows.

If the previous task was still running when it called `schedule()`, i.e., it exhausted its quantum, was preempted, or yielded the processor, then the ELSC scheduler inserts the task into the run queue. This step is important because tasks are removed from their run queue lists when they are executing and need to be put back on the run queue. Even if the task has yielded, it will be treated properly in the search loop. So we insert the task in the table now lest we lose track of it. Also, by re-inserting the previous task here, we do not need to treat it as a special case when evaluating the goodness of tasks. Next, just as the current scheduler, ELSC moves exhausted `SCHED_RR` tasks to the ends of their lists.

The next step determines whether we need to recalculate counters. If the top pointer is zero, then there are no runnable tasks in the table with a non-zero counter value; either they all have zero counter values or there are no tasks in the run queue. If the `next_top` pointer is non-zero, then there are runnable tasks in the table with zero counter values, so the scheduler recalculates the counter values for every task in the system. If, however, the `next_top` pointer is zero, then the table is completely empty and there are no tasks to run, so we schedule the idle task and skip the rest of the decision process.

If the `top_pointer` is non-zero, the list pointed to by it is guaranteed to have at least one non-zero counter task in it, so we start our search at the top list. The ELSC search loop attempts to emulate the `goodness()` calculation used by the current scheduler. Starting with the first task in the list, ELSC checks to see if the task is still running on another CPU. If so, we shouldn't schedule it. If all tasks in the list are eliminated by this check, then we consider the next populated list and try again.⁴ Next, we check to see if the task has a zero counter value. If we find such a task, then the rest of the list is either empty or unusable, so we break out of the search loop. If, however, the task we are considering has a non-zero counter value, then we evaluate its goodness. The process of selecting a task from the highest list is described below.

If the task has just yielded its processor, we will run it only if we cannot find another task on the list. This policy is slightly different than the current scheduler, which considers a yielded task to have a goodness value of zero. From this point, the task's utility is evaluated just like `goodness()`. Bonuses are given for having the same memory map or running on the same processor. In the uni-processor case, if a memory map match is found, then we break out of the search loop and run the task right away because we won't find another task with a greater bonus.

When we finish examining a task, we mark it to be scheduled next if it has the highest utility seen so far. Then we select the next task in the list and repeat the process. In the worst case, every task in the run queue is placed in the same priority list (and ELSC performance can be no better than the current scheduler). So we limit the number of tasks examined in each list to a number, currently set to be half the number of processors in the system plus five, which is intended to be large enough to find tasks with adequate bonuses on SMP systems, yet still limit the search to a reasonable number of tasks. Not considering the rest of the list shouldn't be a problem, as all tasks in the list have about the same static goodness.

For real-time tasks, the search is actually much simpler. Again, we examine only the first few tasks and don't look at those currently running on other processors. But instead of worrying about yielded processes and bonuses, we simply run the task with the highest `rt_priority` value.

After deciding which task to run next, the ELSC scheduler manually removes the task from its list (i.e., doesn't use `del_from_runqueue()`) and sets run queue node's `prev` pointer to `NULL`. This indicates

⁴ This can only happen on SMP systems.

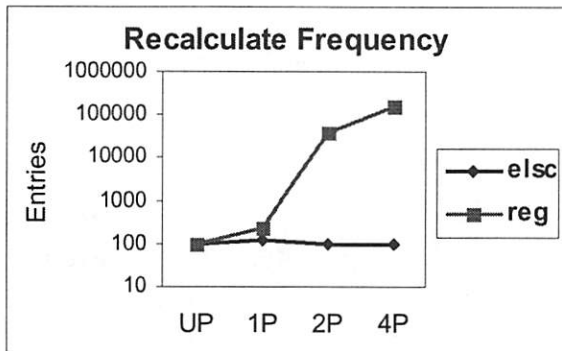


Figure 2: The number of times (on a log scale) that each scheduler enters the recalculate loop during a typical run of the VolanoMark benchmark.

that the task is “on the run queue”, even though it is not currently in a list. Finally, if the previous task had yielded the processor, then the ELSC scheduler clears the `SCHED_YIELD` bit to give the task a better chance in future calls to `schedule()`.

We mentioned before that one of our design goals was to make the ELSC scheduler behave as much like the current scheduler as possible. At this point, we describe how the ELSC scheduler behaves differently. First, the ELSC scheduler tries to limit its search to one list in its table. Therefore, it may choose a task in its highest priority list that doesn’t receive any bonuses for processor affinity or memory map. In this case, it is possible that a task residing in the second highest priority list, which would receive these bonuses and have had a higher `goodness()` value than the chosen task, is not run. We decided this behavioral difference is acceptable because the difference between the `goodness()` values of the two tasks is small enough to ignore.

The other difference in behavior is one that avoids an undesirable characteristic of the current scheduler. Currently, if a task enters the scheduler because it is yielding the processor and no other tasks can be scheduled, then the scheduler enters a loop to recalculate the `counter` value for all tasks in the system. In this situation, the ELSC scheduler runs the previous task again if it does not have a zero `counter` value. Figure 2 illustrates how many times each scheduler recalculates during a typical VolanoMark run on uni-processor and one, two and four processor SMP machines.

6. Experiments

The ELSC scheduler meets the first three of our four design goals. The design changes are kept local, the solution is simple, and it behaves very much like the

Scheduler	Time to Complete Compilation
Current - UP	6:41.41
ELSC - UP	6:38.68
Current - 2P	3:40.38
ELSC - 2P	3:40.36

Table 2: Average time taken to complete a full compile of the Linux kernel.

current scheduler. The final goal of this project is to make the ELSC scheduler perform as well as the current scheduler in lighter desktop situations while scaling gracefully under heavy loads. We used two tests to determine whether we reached this goal. The first is a simple test that measures the time it takes to compile the Linux kernel. This test is meant to compare scheduler performance for light loads. The second test is the VolanoMark benchmark, described earlier. While VolanoMark may not be representative of a typical workload, it does simulate the behavior of a commercially available application. We use it in this analysis as a stress test for the two schedulers.

We compiled the Linux kernel three times on each of the schedulers, configured to run as uni-processor⁵ and two-processor kernels. We ran the test on an IBM Netfinity 5500 with dual Pentium II processors. The kernel version was 2.3.99-pre4 with our ELSC modifications. To run the test, we set up a shell script that would first build a kernel and then run “make clean”. This step was intended to reduce the variance in measurement due to file system performance by pulling as much information as possible into the L1 and L2 caches. Then we use the `bash “time”` command to run the “make -j4 bzImage” command. Table 2 shows the average results given by the time command.

Our confidence in these measurements is very high as the test was run multiple times and results never deviated from the mean by more than 4 hundredths of a second. For all practical purposes, the hundredths of a second reported in Table 2 are insignificant. In the two-processor case the ELSC scheduler barely edges the current scheduler by an insignificant couple hundredths of a second. In the uni-processor case the ELSC scheduler has a distinct advantage. We believe

⁵ In these experiments, uni-processor kernels are compiled without SMP enabled, eliminating its overhead. One-processor kernels are compiled with SMP enabled but use only one processor.

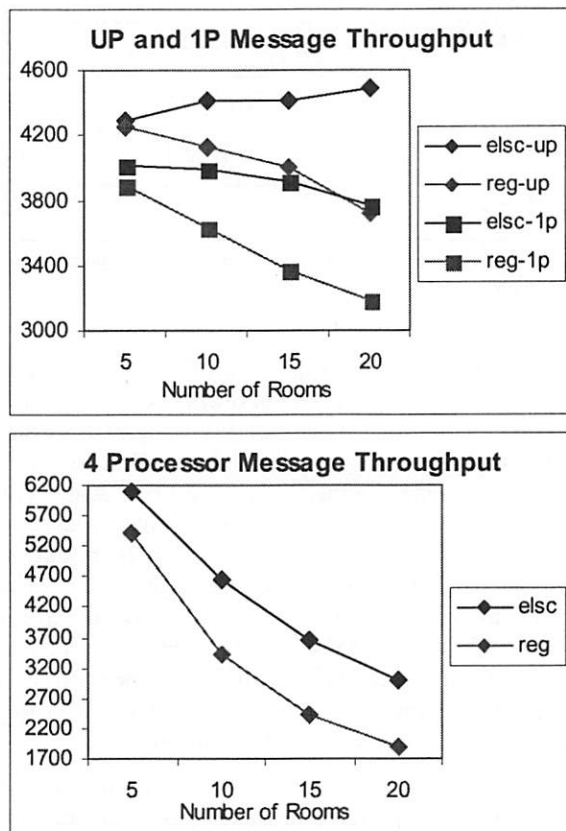


Figure 3: Throughput in messages per second for VolanoMark runs on 6 different scheduler configurations. The Y-scale is adjusted on the second graph to fit all data points.

this is due to the shortcut in the ELSC search loop for the uni-processor scheduler, which ends the search as soon as a memory map match is found.

The VolanoMark benchmark test is more complicated. We ran VolanoMark in loopback mode, which simulates both the clients and servers for the Java chat rooms on the same machine. In loopback mode, communication between clients and servers does not travel across a network. In the exchange of messages between clients and servers, each must have time on the CPU to send and receive it's messages in order to let the other do the same. This type of message exchanging application forces many entries into the scheduler. As suggested by the VolanoMark run rules, we ran the benchmark 11 times for each system configuration and discarded the first run due to its variant startup costs.

We ran VolanoMark with both schedulers configured as uni-processor, one, two and four processor SMP kernels. For each of these configurations, VolanoMark was configured to simulate 5, 10, 15 and 20 rooms, each with 20 simulated users exchanging 100 messages. Each simulated user creates two threads, so each room

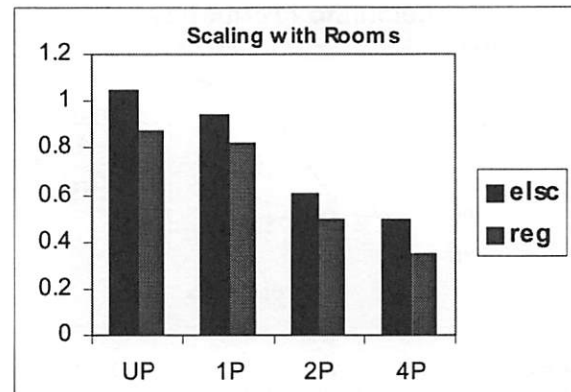


Figure 4: Shows how each scheduler scales from 5 rooms to 20 rooms on various processor configurations. The height of the bar represents the scaling factor (20-room-throughput / 5-room-throughput).

creates a total of 80 threads. It is easy to see that even at 5 rooms the VolanoMark benchmark puts considerable stress on a system. While running VolanoMark, we also collected statistics about what the scheduler was doing and exposed them through the proc file system. The overhead of collecting these statistics exists in both schedulers and in both cases is negligible. The machine used for the VolanoMark runs was an IBM Netfinity 7000 with 4 Pentium II xeon processors.

The metric reported by VolanoMark is message throughput, which we can use as a measure of both performance and scalability. Using the bare results from the VolanoMark runs, we can compare how each of the two schedulers behaves in different configurations. Figure 3 illustrates the performance gains given by the ELSC scheduler.

Figure 4 gives a different interpretation of the same data. To obtain some measure of how well each scheduler scales when faced with a large number of tasks, we can use the 5-room trials as a base measurement and see how performance is altered when the number of threads is increased in the 20-room trials. The number charted in Figure 4 is simply the message throughput achieved in the 20-room trials divided by the throughput achieved in the 5-room trials. As the figure indicates, the ELSC scheduler clearly scales to more threads better than the current scheduler.

But these numbers do not paint the whole picture. We want to understand why the ELSC scheduler scales so much better than the current scheduler and verify that these results are not a fluke. So we collected additional statistics on the schedulers while we ran the VolanoMark tests.

The first statistic that jumps out is the number of cycles spent per entry into the scheduler. For the ELSC

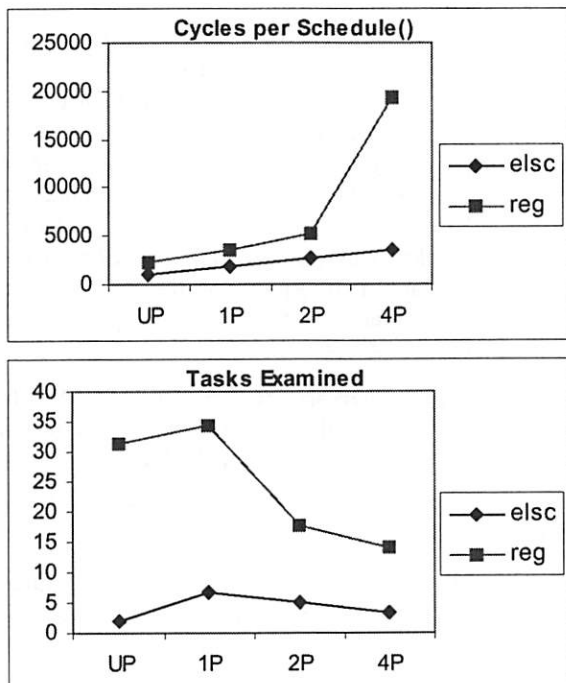


Figure 5: The first chart shows the number of cycles that are spent each time the system enters the scheduler. The second chart shows how many tasks are examined by the scheduler each time it is called.

scheduler, this number is significantly lower than the current scheduler, proving the ELSC scheduler really does spend less time in the scheduler. The explanation is that because ELSC with its table-based approach to scheduling examines far fewer tasks on each entry into the scheduler, as demonstrated by Figure 5.

The ELSC scheduler is not without fault. Although most of the statistics we collected indicate that the ELSC scheduler is faster and better, two of them show the opposite. One of the adverse affects of a table-based scheme is an increase in the number of calls to `schedule()` when running on a machine with more than one processor. As demonstrated by Figure 6, there is a strong correlation with how many times a task is selected without having the processor affinity bonus. These measurements suggest the ELSC scheduler is not choosing the absolute best task in multiprocessor machines. We suspect that this is related to the fact that the ELSC scheduler finds the most suitable task in the highest populated class of static priorities. Thus, some tasks that might have higher `goodness()` values when the processor affinity bonus is added, but reside in lower static classes, may not be considered.

Although the VolanoMark benchmark creates many threads with the same memory map, we do not believe this fact significantly influences the behavior of either scheduler. The only possible difference between the two schedulers would be similar to the passing over of

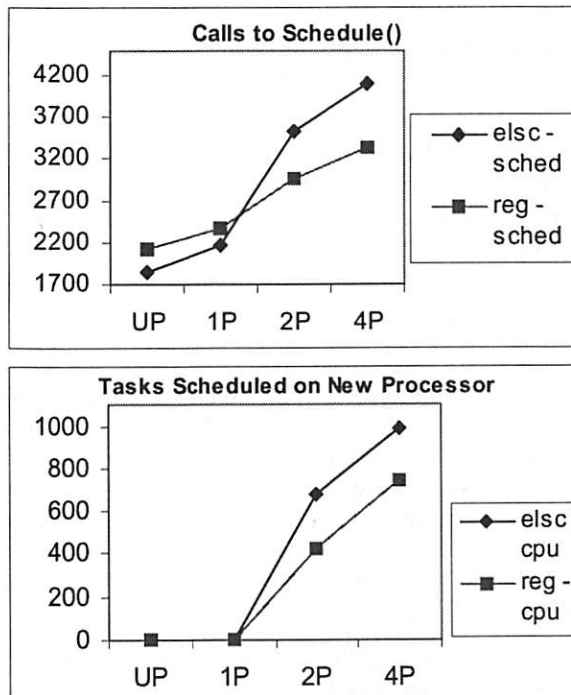


Figure 6: The first chart shows how many times (in thousands) the system enters the `schedule()` function call in an average 10-room VolanoMark simulation. The second chart shows how many times the scheduler chooses a task to run on a different processor than it ran before.

lower classes of static priorities that happens when running on multiple CPU's. Of course, if a task is inserted into a lower priority list, then adding the one point bonus for sharing a memory map with the previous task cannot raise its `goodness` value enough to be greater than any of the tasks in the highest class.

7. Evaluation

An increasing number of organizations continue to evaluate, test, and use the Linux operating system. Although Linux does many things well, we have shown the current scheduler has shortcomings in its design and implementation. When confronted with a large number of tasks, overall system performance declines rapidly. This behavior is unacceptable for large-scale enterprise environments.

We set out to improve the Linux scheduler's scalability, preferring modifications that do not change desktop performance and maintain existing scheduler abstractions, yet scale well when presented with a large number of tasks. We have shown that it's possible to improve the Linux scheduler without introducing a lot of overhead. Though the ELSC scheduler does not always select the best task available on machines with more than one processor, we have demonstrated that the

ELSC scheduler satisfies our goals for both a small and large number of ready tasks and offers a viable alternative to the current Linux scheduler.

The ELSC scheduler is an open source contribution and is freely available for use and modification. The current version of the ELSC patch can be downloaded from www.citi.umich.edu/projects/linux-scalability/patches/

8. Future Work

In the future, we would like to see how the ELSC scheduler performs in other multithreaded environments. One such example is a web server running Apache. Would we see the same performance gains we saw while running VolanoMark, or does something other than the scheduler cause primary bottlenecks in these systems? Would the ELSC scheduler be more effective in increasing throughput or decreasing the latency of an Apache web server?

The focus of the ELSC design is to reduce the time spent looking for a task to schedule. We would also like to find ways to allow the scheduler to make greater use of multiple CPUs and examine the effects of modifying the goodness metric. Is Linux considering everything it ought in its scheduling decisions? Do we care about processor affinity after many other tasks have run on the given processor? Can we construct a scheduler that spends less time waiting for spin locks and more time scheduling tasks?

We are also interested in exploring alternative scheduler designs. The table-based design of the ELSC scheduler is one approach; many other possibilities exist, such as sorting tasks by static goodness within heaps for each processor and address space. One could choose the absolute best task available simply by examining the top of each heap. Or perhaps a multi-priority-queue solution would be more beneficial to help the scheduler scale to multiple processors well.

9. Acknowledgments

We thank Ray Bryant and Bill Hartner of IBM, Chuck Lever of Network Appliance and Brian Noble of the University of Michigan for their guidance and assistance; Dr. Charles Antonelli and Professor Gary Tyson for providing hardware used in development at the University of Michigan; the Linux Technology Center at IBM for allowing the use of equipment at IBM for development and testing; and Chris King, Scott Lathrop and Paul Moore of the University of Michigan for their contributions to the initial development of the ELSC scheduler.

We thank the anonymous reviewers for their helpful comments, and our shepherd, Ted Faber, whose insight and suggestions were particularly valuable.

This work was partially supported by Dell, Intel, and the Sun-Netscape Alliance.

We also thank all the past, present, and future developers of Linux for their skilled and selfless contributions.

10. References

- [1] Atlas, A. "Design and implementation of statistical rate monotonic scheduling in KURT Linux." *Proceedings 20th IEEE Real-Time Systems Symposium*. Phoenix, AZ, December 1999.
- [2] Bryant, Ray and Hartner, Bill. "Java, Threads, and Scheduling in Linux." IBM Linux Technology Center, IBM Software Group. <http://www-4.ibm.com/software/developer/library/java2>
- [3] Carr, John. "AS/400 Leads the league in Java performance." *JavaWorld*, 11 August 2000.
- [4] Daggett, Dawn and Gillen, Al and Kusnetzky, Dan. "Linux Overtakes NetWare for the Market's Number 2 Position." *International Data corporation – Press Release*, 24 July 2000. <http://www.idc.com/software/press/PR/SW072400PR.stm>
- [5] Gooch, Richard. "Linux Scheduler Benchmark Results." 30 September 1998. <http://www.atnf.csiro.au/~rgooch/benchmarks/linux-scheduler.html>
<ftp://ftp.atnf.csiro.au/pub/people/rgooch/linux/kernel-patches/v2.2/rtqueue-patch-current.gz>
- [6] Lohr, Steve. "IBM goes countercultural with Linux." *The New York Times On The Web*, 20 March 2000. <http://www.nytimes.com/library/tech/00/03/biztech/articles/20soft.html>
- [7] Neffenger, John. "The Volano Report." *Volano LLC*, 24 March 2000. <http://www.volano.com/report000324.html>
- [8] Shankland, Stephen. "AOL releases open-source software." *Cnet News*, 9 July 1999. <http://www.canada.cnet.com/news/0-1005-200-344644.html>
- [9] Wang, Y.C. "Implementing a general real-time scheduling framework in the RED-Linux real-time kernel." *Proceedings 20th IEEE Real-Time Systems Symposium*. Los Alamitos, CA, 1999. 246-55
- [10] Woodard, B. "Building an enterprise printing system." *Proceedings of the Twelfth Systems Administration Conference (LISA XII)*. USENIX Association, Berkeley, CA. 1998, 219-28.

A Universal Dynamic Trace for Linux and other Operating Systems

Richard Moore - IBM, Linux Technology Centre - richardj_moore@uk.ibm.com

Abstract

Dynamic Probes (DProbes) from IBM [*] is a generic and pervasive system debugging facility that will operate under the most extreme software conditions with minimal system disruption. It permits debugging of some of the most difficult types of software problem especially those encountered in a production environment that will not readily re-create. It is also an invaluable aid for the developer who has to debug parts of the operating system inaccessible to other technologies. **DProbes** is a front-end enabler for other debugging technologies, such as crash and core dumps and kernel/user debuggers. It is designed to operate with minimal dependence on the operating system, which affords it the possibility of being ported to other operating systems, especially **UNIX** [**] variants, but not limited to **UNIX** as it originated conceptually from **Dynamic Trace** under **OS/2** [*]. This paper describes the latest developments of the **DProbes** project in particular its use as a tracing tool with the **Linux Trace Toolkit** project from **Opersys** [**]. System dependencies are discussed with an emphasis on portability to other **Linux** H/W platforms as well as other operating systems.

1. Introduction

Dynamic Probes (DProbes) for Linux <1> was first released in August 2000 and presented at the Annual Linux Showcase in October 2000 <2>. The original functionality was essentially that of an automated kernel debugger. Since then DProbes has been extended considerably. It now interfaces with a number of external debugging agents, for example: The Kernel Debugger <3> and Kernel Crash Dump <4> facilities from Silicon Graphics Inc. (SGI) [**]; the standard user-space core dump and syslog facilities within Linux and also the Linux Trace Toolkit <5> from Opersys.

The major topics discussed in this paper are:

- Detailed implementation aspects of DProbes that relate to its use as an agent for trace instrumentation under the Linux operating system running on the Intel 32-bit architecture (IA32) <8>.
- Portability considerations across other operating systems running under the Intel 32-bit architecture in particular UNIX-like operating systems.
- Portability to other processor architectures.

Essentially DProbes has become a driver or **enabler for other debugging technologies**. Its enabling capability derives from the following key characteristics:

1. There is a mechanism for intercepting execution at arbitrary code locations - this is the **probepoint** mechanism.
2. Each probepoint has an associated **probe handler** that allows specific actions to be taken. This is implemented using a low-level Reverse Polish Notation (RPN) language that gives access to kernel and user space memory and to the processor's registers^[1].
3. A probe handler terminates in one of three ways:
 - i. By returning to the probed code seamlessly.
 - ii. By returning to the probed code via a logging daemon. A temporary logging buffer is made available for this purpose. This characteristic is exploited to provide a means of instrumenting a module with **tracepoints**.
 - iii. By transferring control to an external debugging facility having first removed the probepoint. Whether or not the original code will continue execution is a function of the external facility.

The efficacy of DProbes is further enhanced by the following three design criteria:

1. There is no required interactive user interface for the probe handler^[2]. This is intentional - it minimizes the dependency of the probe handler on system interfaces and resources. Thus the probe handler is designed to run as a self-contained interrupt handler. The RPN command interpreter provides recovery from potential fatal errors without reference to operating system facilities. This criterion gives DProbes its **universality** since there are very few restrictions on where a probepoint may be placed and when the probe handler may execute. In fact, probepoints are only restricted from being placed in the code path of the probe handler. If such a probepoint were to be defined, DProbes would detect it and silently remove it. Probepoints may therefore be placed in code that runs at task time, interrupt time or during a context switch.

2. The second design criterion was to align a probepoint with a module rather than a storage location. Note that the watchpoint extension, which is described under [7. Dynamic Probes Recent Extensions](#), deviates from this criterion for reasons explained thereunder. By aligning a probepoint with a module, or to be more precise an offset into a module, the probe becomes independent of incidental circumstances that relate to a module's installation in memory and also the processes under which that module is executing. Thus, code that is shared between processes at different virtual addresses (for example a Linux shared library) has location-independent and process-independent probe definitions. This criterion gives DProbes its **independence**, since it makes it possible to describe a probe:
 - i. Independently of an operating system's implementation of module management (clearly the internal implementation needs to understand this);
 - ii. In canonical terms that relate to a programmer's view of his/her module which are: independent of whether a module is loaded at the time the probe is defined; and independent of any particular process under which that module executes.
3. Probepoints are *inserted* into module code paths without the need for source code modifications to that module. Furthermore they may be inserted into any loaded and running code (kernel or user space) or code that is paged out or modules not yet loaded. This mechanism has been described in <2>. In summary, the instruction at the probe location is overlaid with a trapping instruction - under Intel 32-bit architecture (IA32) the **int3** instruction is chosen. The original instruction is either single-stepped or emulated after the probe handler executes. This particular criterion gives DProbes its **dynamic** characteristic. The dynamism refers to its ability to instrument a module with probes *on the fly* so to speak. Thus there is absolutely no performance penalty when a probepoint is inactive.

These characteristics therefore provide an elementary universal dynamic tracing capability, which have been further extended in both universality and dynamism by recent enhancements - see the section: [7. Dynamic Probes Recent Extensions](#) for details.

2. DProbes as a Tracing Mechanism

This section discusses the implementation details of DProbes as a tracing mechanism in detail. We describe first the internal mechanism of the Dynamic Probes Event Handler (DPEH) that enables it to be used as a tracing agent.

DPEH internal details:

DProbes provides various working storage elements for use by the probe handler:

1. Local variable array.
2. Global variable array.
3. A per-processor log buffer.

The latter is intended for use as a staging area for building a trace or log record to be passed synchronously to a logging or tracing daemon external to DProbes. One log buffer is permanently allocated per processor but the data in each buffer persists only for the duration that a probe handler is active on its respective processor. The RPN command interpreter maintains an internal pointer to the next available location in the buffer, which is reset to the beginning of the buffer on entry to the probe handler. Data is thus always accumulated monotonically and discarded on exit from the probe handler.

The buffer is populated using the **log** class of RPN instructions. These are defined in two categories: those that copy data directly from the RPN stack and those that use the RPN stack to specify data to be copied from system memory.

The direct category comprises three instructions in the IA32 implementation:

log b,<n>	Log byte
log w,<n>	Log word
log d,<n>	Log double-word

Pop <n> elements from the RPN stack and from each, copy the least significant byte (8-bit integer), word (16-bit integer) or double-word to the log buffer.

The indirect category has four members in the IA32 implementation. Each operates by popping an address followed by a length from the RPN stack. Data, for that length, at that address, is copied to the log buffer. However, before data is copied, it is appended with a 3-byte prefix that contains a token byte and a length word. The length refers to the length of data that follows

the prefix and the token byte to type of data. The feature enables:

- a. A trace or log record to contain variable length data such as arrays whose length is determined dynamically.
- b. A formatting utility to operate using a fixed template for variable length data.

The token byte values are defined as follows:

- | | |
|----|---|
| 0 | binary data logged successfully of length specified by the prefix length word. |
| 1 | (ASCII) string data logged successfully of maximum length specified by the prefix length word. The actual length of the data may be less if a terminating NULL byte is encountered within the prefix length. This allows data of arbitrary lengths to be capped especially in cases where string data has been corrupted. |
| -1 | a fault occurred accessing the data using a flat address. The prefix length is set to 4 and the variable data contains only the (flat) address that caused the fault. |
| -2 | a fault occurred accessing the data using an invalid selector. The prefix length is set to 4 and the variable data contains only the selector for the segment generating the fault. |

Note: the latter two tokens may occur in circumstances where the data address was valid. Since the RPN probe handler executes, essentially as an interrupt handler, with minimal access to system facilities it will not be able to recover from otherwise recoverable faults. This is a trade-off between the universality and flexibility of DProbes. See [3. DProbes Event Handler Processing](#) for a further discussion on how such conditions are handled.

Under the IA32 implementation there are 4 RPN instructions use for logging data in memory:

log mrf	Log memory range from flat address.
log mrs	Log memory range from segmented address.

Pop a flat address (**log mrf**) or a 16-bit offset then a 16-bit selector (**log mrs**) from the RPN stack. Then

pop a length. Reserve space for the 3-byte prefix in the log buffer. Copy data from the address specified for the length specified to the log buffer. If a fault is generated then set the prefix token to -1, the length to 4 and store the fault address. If no fault is generated then set the prefix with a 0 token and length of data logged.

log arf	Log ASCII range from flat address.
log ars	Log ASCII range from segmented address.

Pop a flat address (**log arf**) or a 16-bit offset then a 16-bit selector (**log ars**) from the RPN stack. Then pop a length. Reserve space for the 3-byte prefix in the log buffer. Copy data from the address specified up to the length specified or until a NULL terminator byte has been copied. If a fault is generated then set the prefix token to -2, the length to 4 and store the fault address. If no fault is generated then set the prefix with a 1 token and maximum length value popped from the RPN stack.

3. DProbes Event Handler Processing

We turn now to considerations concerning back-end probe event handler processing.

As described in <2>, the DProbes Event Handler (DPEH) needs to execute the original instruction that was replaced with a breakpoint. It does this by single-stepping the original instruction *in situ* with interrupts disabled^[3]. If that instruction faults then we require the operating system to recover and retry the instruction. However, one does not normally wish to have multiple trace records generated for each retry execution of an instruction, especially when that instruction eventually succeeds and will thus appear to the underlying program to have executed only once, and with success. This is achieved by delaying the call to the tracing daemon until after the original instruction has completed single-step. If a fault is generated then the daemon is not called and the log buffer is reset. Furthermore the DPEH reinstates the probepoint - **int3** instruction under IA32 - and resets interrupt status, saved by the processor when the fault was generated, to indicate the status prior to execution of the probepoint. Finally it returns to the system fault handler to allow normal fault processing to occur. If the system retries the faulting instruction it will unwittingly retry the probepoint instruction. The DPEH will therefore be called for each retry. Only on successful execution of the original instruction will the trace daemon be called.

It is a requirement for the probe handler to be re-executed for each retry of a faulting instruction. This is because it is quite possible, in the case of a page fault, that the data causing the instruction to fault is also accessed by the probe handler for copying into the log buffer. Only on successful execution of the original instruction would the trace record in this case be complete.

Some instructions generate faults for non-error reasons, for example IA32 **bounds** instruction. For such instructions it would be desirable to log a trace record despite a fault being generated on single-step. This is now possible by means of the **logonfault** control statement, which is specified in the header of the RPN file. This feature was added recently to DProbes - see [7. Dynamic Probes Recent Extensions](#) below.

4. DPEH Performance Implications

We have made an initial study of the performance overhead of a probepoint. A more comprehensive performance evaluation is future work. The first set of results are quantitative observations made under the Linux 2.2.12 kernel. We also present some qualitative results taken from real-life usage under OS/2. The conclusions from these OS/2 examples indicate that under most conditions the impact of a probepoint is negligible when active. We concern ourselves only with measuring the effect of the active probepoint, since for inactive probepoints there is no alteration to the code path and therefore a zero overhead.

We estimated the overhead of the DPEH using a 90 MHz Pentium[**] processor^[4]. Five experiments were performed:

1. To obtain a base measurement of the time taken to execute a sequence of the following three single cycle instructions in a loop:


```
loop: dec eax
      nop
      jnz loop
```
2. To test a null probe handler with only the **abort**^[5] RPN instruction and with the probe placed on the **dec eax** instruction. Here **dec** is single-stepped by the DPEH.
3. Using the same probe handler but the probe placed on the **nop** instruction. Here **nop** is emulated by the DPEH.

4. Using same probe location as 2 but a single **push eax**^[6] RPN instruction added to the probe handler.
5. The same probe location as 2 but a single **exit**^[7] RPN instruction in the probe handler.

The results were as follows:

1. One iteration of the three-instruction loop averaged 30ns, each instruction approximately 10ns .
2. One iteration of the loop averaged 16µs. Therefore the minimum overhead of the DPEH is approximately 16µs.
3. One iteration of the loop averaged 8µs. Therefore the cost of the DPEH back-end single-step processing accounts for half the overhead per probe.
4. One iteration of the loop averaged 16µs. **push eax** therefore has a negligible effect. One might reasonably assume most register and memory based RPN instructions are of a similar overhead.
5. One iteration of the loop averaged 200µs. Most of this is the cost of a **printk**, which is the default logging method (see [5. The Trace Daemon Interface](#) below for details on how **printk** is invoked).

Taken at face value the minimum overhead of the DPEH appears to be of the order of 10³. This would certainly be a valid perception if a probe were placed in a tight CPU-bound loop. However, in most applications of DProbes the average number of instructions executed between consecutive executions of the same probepoint outweighs any overhead imposed by the DPEH. This is illustrated by the following qualitative results taken from real-life uses of DProbes (actually Dynamic Trace) under OS/2:

1. Tracepoints^[8] on every kernel API entry and exit (circa 500 tracepoints).
The user perception varies from unnoticeable to very slight depending on work load. The system is useable and performs within acceptable norms.
2. Tracepoints on entry and exit to the process context switching code with page table data logged on entry and exit.
No noticeable overhead.

3. Tracepoints on every OS/2 Presentation Manager [*] API entry and exit (circa 500 tracepoints).
A noticeable slowing of GUI response. The GUI was useable.
4. Tracepoints on entry to page allocation, page de-allocation and page fault handling routines in the OS/2 page manager.
No noticeable overhead.
5. Tracepoints on 4000 internal kernel routines.
Very noticeable, however the system was still useable.

Conclusions

While the cost of a probe is not cheap it can be considerably reduced by placing the probe on an emulated instruction such as **nop**. It can also be reduced by judicious use of logging by employing conditional logic in the probe handler to avoid unnecessary log events. But foremost, the practical use of DProbes finds probepoints being placed in code paths with a relatively long mean time to iterate. Under these circumstances the overhead is negligible.

5. The Trace Daemon Interface

The generic requirements for a trace daemon interface are:

1. To provide a logging API capable of being called from kernel space, while interrupts are disabled, from both a task-time and interrupt-time context.
2. To allow binary data of an arbitrary length to be logged and identified as originating from DProbes.

A number of candidates satisfy these requirements. The default behavior is to invoke the **klog** daemon via **printk**. Other options include directing output through a dedicated asynchronous communications port (**com1** or **com2**). Strictly speaking, using a communications port doesn't necessarily invoke a daemon unless one thinks of the monitoring system connected to the system running DProbes as a daemon. And finally, a local tracing daemon can be invoked to record the log buffer. Use of this option requires a degree of conformance between both DProbes and the tracing facility. We have chosen to use the Linux Trace Toolkit from Opersys <5> as an initial implementation. We will describe a little later a generic interface that is possible to implement by using the Generalised Kernel Hook Interface <6> mechanism.

Logging to the Communications ports or klog:

Logging to both the **com1** and **com2** communications ports and **klog** involves converting the log data to an ASCII string of pairs of hexadecimal characters and outputting that to the respective medium. Prior to this we format a record header that contains both constant information and some optional entities that are common to all tracepoints. The most generalised form of the trace header template is as follows:

```
"DProbes(%d,%d) cpu=%d name=%s pid=%d
uid=%d cs=%x eip=%08lx ss=%x esp=%08lx
tsc=%08lx:%08lx\n"
```

Other than the **DProbes(...)** text item, every other item is optionally present. All but **cpu** are selectable by the user through parameter switched to the **dprobes** command; **cpu** is activated automatically when DProbes is run from a multi-processor system.

The meaning of each constituent header item is as follows:

DProbes(%d,%d)

Displays the major and minor code that identifies the probepoint. Each probepoint has assigned a major and minor identifier. These are not required to be unique, but by convention are chosen to indicate a unique type of probe, for example the exit point(s) of a particular routine. Major and minor codes are intended to be used by a generalised formatter to identify a unique formatting template. See [6. Trace Formatting Interface](#) below

cpu=%d

Displays the processor id on which the probe was executed. This is suppressed on uniprocessor systems and always displayed on multi-processor systems.

name=%s

Displays the process name taken from the current task structure when the probe was executed.

pid=%d

Displays the process id taken from the current task structure when the probe was executed.

uid=%d

Displays the user id name taken from the current task structure when the probe was executed.

cs=%x eip=%08lx

Displays the CS and EIP registers at the probe location. This is sometimes useful in distinguishing individuals of a group of similarly formatted and therefore identical major and minor coded probes. For example, multiple return points from a function.

ss=%x esp=%08lx

Displays the SS and ESP register values when the probe was executed. This can give an indication of the nesting level of a subroutine.

tsc=%08lx:%08lx

Displays the high resolution processor time-stamp counter in seconds and micro-seconds.

The remaining data is output as an ASCII string of hexadecimal characters.

Logging to a Trace Daemon

We chose to use the Linux Trace Toolkit (LTT) <5> from Opsys as the trace recording daemon. It provides the usual post-processing, formatting and analysis features as well as a daemon that manages the a kernel space trace buffer and a mechanism for off-loading the trace buffer to disk. But most importantly, the Linux Trace Toolkit was conceived as a kernel based static^[9] tracing mechanism, capable for having tracepoints placed in both interrupt handlers and code that runs with interrupts disabled. In other words the conditions under which the DPEH executes. We were able extend the Linux Trace Toolkit to provide an kernel programming interface that allows data to be logged of a an arbitrary length.

The KPI interface to Linux Trace Toolkit's Raw Data interface is show below:

```
struct trace_raw {
    uint32_t id; /* Event ID */
    uint32_t DataSize; /* Size of data
                      recorded by event */
    void* Data; /* Data recorded by
                event */
}

#define TRACE_RAW(ID, LEN, DATA) \
do { \
    struct trace_raw raw_event; \
    raw_event.id = id; \
    raw_event.DataSize = LEN; \
    raw_event.Data = DATA; \
}
```

event

This an event identifier defined by LTT. It signifies a binary data record, the format of which is undisclosed to LTT.

id

This a module identifier returned by LTT when tracepoints for a given module are activated. DProbes calls the LTT **trace_create_event()** routine when it inserts tracepoint for a given module. This enables LTT to correlate events with a module for the purposes of event analysis. Note: DProbes will call LTT **trace_destroy_event()** routine when tracepoints for a module are removed.

DataSize

This is the overall size of the trace record (flags + + log buffer content).

Data

This is a pointer to the trace record (flags + header + log buffer content).

The logged data is further structured with a header followed by the data from the log buffer. The header comprises a flag double-word followed by one or more binary data items concatenated together. The presence of an item is signified by its corresponding flag bit being set. The following table shows the format of each header item and its corresponding flag setting in the order they appear in the header:

#	flag	type	description
1	0x0001	uint32	major
2	0x0002	uint32	minor
3	0x0004	uint32	cpu
4	0x0008	uint32	pid
5	0x0010	uint32	uid
6	0x0020	uint32	cs
7	0x0040	uint32	eip
8	0x0080	uint32	ss
9	0x0100	uint32	esp
10	0x0200	uint64	tsc
11	0x0400	string	process name

This implementation is specific to LTT, but may be readily adapted to other daemons either by requiring that they support the three interfaces for creating, destroying and logging an event.

Generalised Kernel Hook Interface

The disadvantage of the implementation just described is that DProbes needs to be built for use with LTT and LTT needs to be present in the system before DProbes loads in order to resolve the external references to the three interfaces. We can avoid this problem by using the Generalised Kernel Hook Interface <6> to define hook exit points within DProbes for the three interfaces. An arbitrary trace daemon would register and arm exit routines for these three hooks when the daemon loads or is instructed to do so. Because the state of activation of a GKHI hook is transparent to DProbes, it would execute code paths that call the three interfaces (now hooks) without regard to whether a recipient daemon had armed them. The equivalent hook exit points for each of the three API calls is coded as follows:

```
trace_event(event, &event_struct);
GKHOOK_2VAR(GKHOOK_DPROBES_LOG_EVENT, event, &event_struct);

event=trace_create_event(name, format, desc);
GKHOOK_4VAR(GKHOOK_DPROBES_CREATE_EVENT, &event, &name, &format, &desc);

rc=trace_destroy_event(event);
GKHOOK_2VAR(GKHOOK_DPROBES_DESTROY_EVENT, &rc, event);
```

DProbes would notify GKHI of the existence of these three hooks during initialisation by calling **GKH_identify**.

6. Trace Formatting Interface

Clearly, a hexadecimal format for the trace record is not the most user friendly. Therefore we have proposed a formatting utility in the form of a set of shared library routines that may be called to format individual trace records. The unformatted binary trace record is passed to the formatter and a pointer to the formatted trace record is returned.

The formatter uses text templates with place-holders to format the raw data. For efficiency, templates are cached in memory. The formatting library provides two additional subroutine calls:

1. **initialise**, where essentially the template directory file is opened, loaded and closed.
2. **terminate**, where any cached templates are freed.

Note: these two interfaces may be called sequentially, in reverse order to allow templates to be re-read from disk following an update.

Formatting Template Structure

The template syntax is an extension and simplification of that employed by the OS/2 Trace Formatter, which is a natural thing to do since Dynamic Probes also owes its origin to OS/2's Dynamic Trace facility <7>. This scheme is based on a **printf**-like formatting template. But as discussed below, we have a requirement to format arrays and binary data (essentially an array of bytes) whose number of elements is only determined at the time a trace record is created. This requirement necessitates deviation from a simple **printf** template.

By convention a unique major code is assigned per module. Each unique trace record format for a module is assigned a unique minor code within the major code. This allows us to employ one formatting template file per major code. A template directory is employed to cross-reference major code to template file name. The template file needs only to identify minor code to delimit each template, however, for sanity purposes the major code is coded at the head of the file. Comments are allowed using c-style comment syntax.

Each formatting statement is of the form
keyword=<value>

Numeric values are allowed to be expressed in decimal and hexadecimal using c-notation.

Strings are quoted using c-notation.

The first statement of the file is:
major=<major code>

Subsequent statements will follow the format:

```
minor=<minor code>[,]  
desc=<"descriptive header text">[,]  
fmt=<"template 1">[,]  
fmt=<"template 2">[,]
```

•
•
•

End of file or the next **minor** keyword delimits the end of the previous template.

The **desc** statement serves to provide a static text description of the trace event.

Major, **minor** and **desc** are mandatory, **fmt** is optional, however if **minor** is omitted then only default formatting will be performed. The data will be treated as binary and formatted in dump format displaying offsets, hexadecimal and ASCII.

The **fmt** statements are used to supply template information for formatting user data in the trace record.

In general any alphanumeric character found in the **fmt** statement is treated as literal text and copied directly to the output buffer. Escape control characters **\n** and **\t** are supported. In general the last pair of characters in a sequence of **fmt** statements will be **\n**, however the formatter will always generate an additional new-line at the end of a new trace record.

Multiple **fmt** statements for the same minor code are concatenated by the formatter, so the user must supply necessary spacing and new-line characters if the formatted data is to span more than one line. Place holders for data to be extracted and formatted within the template is signified by a sequence that is prefixed with a **%** character. Multi-byte control sequences are terminated by any non-numeric character, since in a multi-byte control sequence the trailing characters are numeric.

The following control sequences may be specified:

%<n>c - format **n**-bytes as an ASCII characters. If the character is in the range 0x20-0x7f then format the ASCII equivalent character, otherwise substitute a period.

%<n>d - format and **n**-byte decimal integer with leading zeros removed.

%<n>f - format an **n**-byte floating point numeric with leading zeros removed.

%<n>i - skip **n** bytes in the unformatted data buffer.

%p - skip the three-byte prefix for variable length data, see the description of the logging RPN commands under 2. DProbes as a Tracing Mechanism above. This is used in combination with most other controls by placing them after **p**. Controls **u** and **r** are excluded from use with **p**.

%r - skip the three byte prefix for variable length data, but use it as a repetition control, see below. This is used with other controls or a complex expression following.

%s - format an ASCII string up to the length specified by the **%p** prefix, or until a null terminator is

encountered. If **%p** is not specified then **%s** formats a string until a null is encountered.

%<n>u - format an **n**-byte unsigned decimal integer with leading zeros removed.

%<n>x - format and **n**-byte hexadecimal integer including leading zeros.

%z - format the remainder of the trace record in dump format (offset, 0x20 hexadecimal bytes separated by spaces and ASCII equivalent for each 0x20 bytes, repeated for each 0x20 bytes - one per line).

```
+00000000 21 22 23 24 25 26 27 28 20 c4 a8
fe ae ef ff bb *abcdefgh .....*
+00000020 21 22 23 24 25 26 27 28 20 c4 a8
fe ae ef ff bb *abcdefgh .....*
+00000040 21 22 23 24 25 26
*abcdef*
```

(- begins a complex expression - see below

) - ends a complex expression - see below

Where a **<n>** qualifier is allowed then its omission defaults to 1.

Processing the 3-byte prefix

%p causes the formatter to skip over the prefix, noting the code and length. If an error is indicated an error message is formatted.

If **%s** follows **%p** and the code is 0x01 then data is formatted up to the first null character or until the length is exhausted.

If **%s** follows **%p** and the code is 0x00 then data is formatted up to the first null character and any remaining data up to the value of the length is skipped.

If any other control follows **%p** then that data is formatted according to the following control, having skipped the prefix (the error code being checked first).

%p may be combined with any control other than **%r** and **%u**.

%r is used to process the prefix in a similar way to **%p**, except in this case it uses the prefix to repeat the control sequence that follows until data of the length specified by the prefix is formatted. **%r** may be combined with any control though it seldom makes sense to combine it with **%p**, **%s** in simple formatting expressions.

When controls are combined only one % is specified.
For example:
%ps - causes a prefixed string to be processed.

When two data items are to be concatenated then two % signs are needed. For example:

%4us - formats a 4-byte unsigned decimal integer suffixed with a character s, whereas
%4u%s - formats a 4-byte unsigned decimal integer concatenated to a zero terminated string.

%r may be followed by a left parenthesis (to form a complex formatting expression, which is completed with a right parenthesis). This device allows arrays of structures to be formatted. For example an array for which each entry contained two double-words called "function" and "return code" would be formatted using:

%r(function=0x%2x return code=0x%2x\n)

The result would be (for a length value of 12 in the prefix):

Function=0x0000 return code=0x0000
Function=0x0000 return code=0x0003
Function=0x0002 return code=0x0000

A more complex example where the array is a table pointers to strings could be formatted using:

%r(pointer=0x%4x, string='%ps'\n)

The result would be:

pointer=0x801234455, string='this is an example string'
pointer=0x802234455, string='this is another example string'

Within a complex expression the % must be used to prefix groups of controls.
To format a literal %, (or) character then an additional prefix % is required. For example:

%%	results in	%
%(results in	(
%)	results in)

Note: there is scope for extending this scheme to cope with formatting bit masks and conditional formatting and this is something we plan to do.

7. Dynamic Probes Recent Extensions

Since its original release, Dynamic Probes has been enhanced with a number of new features which are relevant to tracing. These are briefly described below:

Watchpoint Probes

This innovation defines a new class of probe that exploits the hardware watchpoint^[10] architecture. Watchpoints are specified by watch-type, which under IA32 may be Read, Write, Execute or IO; and address range. Watchpoints are global and not aligned with any particular module, however symbolic expressions are permitted in the specification of a watchpoint address. This capability gives DProbes its ability to trace memory accesses.

Logonfault

This allows the option of logging the contents of a log buffer whether or not the instruction at the tracepoint generates an exception during single-step. If the operating system retries the instruction then multiple events will be logged. This was introduced to handle two circumstances:

1. where instructions such as **bounds** generate exceptions as part of normal execution and the exception is not subject to seamless recovery by the operating system.
2. when a probe is used for monitoring program efficiency. For example, by logging all attempted executions of an instruction that is capable of generating a page-fault. By this means one may glean an insight into the effects of a particular code path on demand paging.

In both cases it is acceptable log each execution of the probed instruction whether or not it is for recovery purposes.

Probe handler exception handling

This capability allows an RPN probe handler to specify a label from which execution will continue should a fault occur when processing a **log** instruction. The 3-byte prefix is optionally generated with the error code depending on the definition of the exception handler. Interpretation of the RPN probe handler is allowed to continue.

Call Kmod

This allows an open-ended extension to the RPN command set, by providing a hook for which any

kernel module may register. The call **kmod** RPN instruction will give control to the hook exit routine. GKHI is used to implement this interface.

8. Porting Considerations:

Because DProbes relies on few operating system interfaces it is relatively easy to port to other operating systems, especially of the UNIX variety. Furthermore it is structured in a way that enables it to be ported to other architectures besides IA32. IBM is currently working on ports to the zSeries (31-bit and 64-bit) [*] and Intel 64-bit <8> architectures.

Porting to Linux on other processor platforms

The following are the key items to be translated when considering a port to another processor architecture:

Integer size

The global and local variable array element size is set to the integer size (in multiples of 8). So also the element size of the RPN stack. All these dependencies are tied to a single **#define** definition.

RPN instruction set

References to processor registers need to be mapped to the new architecture. Each register push instruction is actually an alias for the single instruction **push r,<n>**. The aliases are implemented by the **dprobes** command from a table that cross-references register to register number.

The push byte, word and double-word set may need to be extended to include a quad-word (64-bit). The will need to be implemented to produce the correct results for the particular endian characteristic of the processor.

It is unlikely that a probe handler written for one architecture would work without modification for another. However this can be addressed by using a high-level language interface for probe handler definitions. This would avoid low-level CPU based constructs and have a good chance of being architecturally independent.^[11]

Probepoint implementation

Probepoints are implemented trapping instruction breakpoints. The processor architecture must provide an instruction that can be stored atomically and will case a privilege-level switch. For example, **SVC 255** serves this purpose for IBM zSeries processors. The interrupt handler for the

breakpoint instruction will need to be hooked by the DPEH.

Single-step

The original instruction at the probepoint needs to be single-stepped. Such a mechanism must therefore exist for use under software control. Use of the hardware watchpoint mechanism may be needed to implement this. Under IBM zSeries, one would use the Program Event Recording (PER) facility. If no inherent single-step capability exists then use of additional breakpoint instructions will be required - this however is an imperfect solution which may prohibit the specification of probepoints on jump or call instructions.

Processor exceptions

All processor exceptions that are generated through normal instruction execution need to be intercepted as part of the single-step back-end processing. Additionally the sequence from breakpoint interrupt through to single-step interrupt needs to be conducted with interrupts disabled in order:

1. to preserve event sequences where an interrupt occurs during the processing of a probe event
2. to avoid difficulties that arise with recursion through the DPEH.

Under Linux for IA32 this required both the exception 1 and exception 3 trap gates to be converted to interrupt gates.

Processor serialisation

Under a multi-processor environment the single-step optionally needs to be executed while other processors suspend execution. If this facility cannot be guaranteed then the **-stopcpus** switch of the **dprobes** command will not be supportable.

Instruction cache serialisation

Because instructions of loaded modules are dynamically altered, serialisation of the instruction pre-fetch cache may need to be performed. Under Linux the **flush_icache** operating system call achieves this.

Watchpoint implementation

This is more complex and difficult to generalise. In the worst case scenario, watchpoint probe support will have to be removed. Otherwise support is a

matter of mapping the watchpoint address and range the processor implementation. The DPEH watchpoint event interface will need to hook the watchpoint interrupt handler. It is likely that a generalised debug register allocation scheme will be needed along with adjustments to context switching to ensure registers used for watchpoints are global to all contexts and can easily co-exist with other uses of watchpoints within the system^[12].

Porting to other operating systems.

There are four key considerations:

Module management

DProbes requires a unique handle by which it can refer to a module while either loaded or on disk. There needs to be a means of correlating a virtual storage address in a given context with a module handle. Under Linux the **inode** serves this purpose.

Page management

Probepoints need to be re-inserted when a module page is brought into memory. Under Linux DProbes achieves this by hooking the **readpage** address. If the paging mechanism is not used make the initial load of a module, as in the case of Linux kernel modules then module load and unload will also need to be hooked.

Symbolic support

To support symbolic expressions the expression-analyser in the **dprobes** command will need to be adapted to process the module format. Under Linux DProbes assumes the ELF format, which is common to many UNIX-like platforms.

Memory management services

Apart from basic allocation and de-allocation functions, DProbes will require a means of aliasing a physical page with a private writeable virtual address to be able to store the breakpoint instructions without causing a fault or a proliferation of privatised pages, which would be the case where a Copy-on-Write page management scheme is implemented.

Fault handling

The DPEH needs to intercept faults relating to access violations before any operating system processing so that they may be *silently* handled by the DPEH RPN command interpreter.

9. Where to obtain DProbes and GKHI:

DProbes, and GKHI are available from the IBM Linux Technology Centre's web page at:
<http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes>

The development team comprises:

Richard J Moore (DProbes Project Lead) -
richardj_moore@uk.ibm.com

Bharata B Rao - rbharata@in.ibm.com

Subodh Soni - ssubodh@in.ibm.com

Vamsikrishna Sangavarapu - rlvamsi@in.ibm.com

Suparna Bhattacharya - bsuparna@in.ibm.com

10. References

- <1> Dynamic Probes is an open-source project distributed freely under the GNU GPL from
<http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes>
- <2> Dynamic Probes and Generalised Kernel Hooks paper published in the USENIX Proceedings of the October 2000 Annual Linux Showcase.
- <3> The SGI [**] Kernel Debugger is an open-source project from Silicon Graphics Inc.. It may be obtained from:
<http://oss.sgi.com/projects/kdb>
- <4> The SGI Kernel Crash Dump is an open-source project from Silicon Graphics Inc.. It may be obtained from:
<http://oss.sgi.com/projects/lkcd>
- <5> The Linux Trace Toolkit is an open-source project from Opersys, Motreal. It may be obtained from:
<http://www.opersys.com/LTT/>
- <6> Generalised Kernel Hooks Interface is an open-source project distributed freely under the GNU GPL from:
<http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes>
- <7> OS/2 Trace facilities are described in the OS/2 Debugging Handbook Volume 3. Order number SBOF 8617 or as an on-line Redbook under order number SG244640.
- <8> IA32 and IA64 are abbreviations for the 32-bit Pentium and 64-bit Itanium processors of the Intel Corporation [**].

11. Trademarks

[*] IBM, OS/2, zSeries, S/390 and Presentation Manager are trademarks of the International Business Machines Corporation in the United States and other countries.

[**] UNIX is a registered trademark of The Open Group in the United States and other countries.

Intel, Pentium and Itanium are trademarks of the Intel Corporation in the United States, other countries, or both.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

12. Notes

[1] An RPN language is used for the following reasons:

- a. it allows a simple abstraction of the processor architecture to be defined to give access to the lowest level resources for minimal overhead.
- b. it provides a basis on which high-level language interfaces can be defined and be largely architecturally independent. Compare this with the Java [**] language and its implementation by a Java Virtual Machine which has an RPN-based virtual machine code.

[2] An interactive interface could always be provided by transferring control to a debugger such as the SGI kernel debugger.

[3] The reasons for this restrictive behavior are described in <2>. In summary this is due to the fact the recursion cannot be tolerated by the DPEH since few system services are available to it, in particular memory allocation. It would be possible to tolerate a finite level of recursion using a DPEH state saving stack independent of the IA32 implemented stack, however, performance and boundary conditions become complications. The latter in particular, since it would be difficult to manifest a consistent behavior to the user.

[4] These experiments were subsequently repeated using an Intel 200MHz Pentium processor. The results were consistent with those obtained earlier using the Intel 90Mhz Pentium processor, being scaled by a factor of approximately 50%.

[5] The **abort** RPN instruction causes probe handler to exit without calling any external logging function.

[6] **push eax** stores the value of the EAX register on the RPN stack. The processing by the interpreter for this instruction similar to that of most of the RPN instruction set.

[7] The **exit** RPN instruction causes the probe handler to exit and for the default external logging function to be called.

[8] A tracepoint is a probepoint used for the purpose of tracing.

[9] Static as opposed to dynamic trace refers here to tracepoints that are hard coded in program source as

opposed to dynamically inserted at run-time. With static trace there is always an overhead even when the tracepoint is inactive.

[10] Watchpoints refer to processor implemented breakpoints that require no code modification. In general they are implemented using special registers and features of the processor. They normally are not confined to monitoring execution but also permit memory references to be monitored. Watchpoints are usually global in nature being specified by virtual or even physical address location under some architectures.

[11] The IBM Dprobes team is working on a current project to implement a high-level language preprocessor for DProbes which generates RPN instructions from a c-like probe definition language.

[12] The IBM DProbes team submitted a Linux kernel patch to the Linux Kernel Mailing List to achieve this for Linux under IA32.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association:

Addison-Wesley	Microsoft Research	Sendmail, Inc.
Kit Cospir	Motorola Australia Software Centre	Smart Storage, Inc.
Earthlink Network	New Riders Publishing	Sun Microsystems, Inc.
Edgix	Nimrod AS	Sybase, Inc.
Interhack Corporation	O'Reilly & Associates Inc.	Syntax, Inc.
Interliant	Raytheon Company	Taos: The Sys Admin Company
Lessing & Partner	Sams Publishing	TechTarget.com
Linux Security, Inc.	The SANS Institute	UUNET Technologies, Inc.
Lucent Technologies		

Supporting Members of SAGE:

Certainty Solutions	Mentor Graphics Corp.	Remedy Corporation
Collective Technologies	Microsoft Research	RIPE NCC
Electric Lightwave, Inc.	Motorola Australia Software Centre	Sams Publishing
ESM Services, Inc.	New Riders Publishing	SysAdmin Magazine
Lessing & Partner	O'Reilly & Associates Inc.	Taos: The Sys Admin Company
Linux Security, Inc.	Raytheon Company	Unix Guru Universe

For more information about membership, conferences, or publications,

see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA

Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-880446-10-3